USENIX USENIX Symposium on Internet Technologies and Systems *Monterey, California* *December 1997*

conference

proceedings

USENIX Symposium on Internet Technologies and Systems Proceedings

Monterey, California
December 8–11, 1997

Sponsored by
The USENIX Association

USENIX®

The Advanced Computing
Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $20 for members and $26 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-91-X

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the USENIX Symposium on Internet Technologies and Systems

December 8-11, 1997
Monterey, California

Conference Organizers

Program Chair

Carl Staelin, *Hewlett-Packard Laboratories*

Program Committee

Mary G. Baker, *Stanford University*
Matt Blaze, *AT&T Labs – Research*
Eric Brewer, *U.C. Berkeley and Inktomi*
Paul De Bra, *Eindhoven University of Technology*
Richard Golding, *Hewlett-Packard Laboratories*
Larry McVoy, *Cobalt MicroServer, Inc.*
Pat Parseghian, *Transmeta Corp.*
Margo Seltzer, *Harvard University*
Doug Tygar, *Carnegie Mellon University*

External Reviewers

Qiming Chen, *Hewlett-Packard Laboratories*
Matt Franklin, *AT&T Bell Labs*
Bill Serra, *Hewlett-Packard Laboratories*
Mirjana Spasojevic, *Hewlett-Packard Laboratories*

The USENIX Association Staff

Table of Contents

USENIX Symposium on Internet Technologies and Systems

December 8-11, 1997
Monterey, California

Preface

Welcome to the USENIX Symposium on Internet Technologies and Systems!

I am very excited about this symposium. The program addresses some of the most pressing problems with the Internet today, such as performance and security. The last three years have seen the rapid growth of the Internet and its transformation from an arcane technology available to a small number of researchers to a widely available infrastructure used by companies and consumers to conduct ordinary business. The Web has visibly affected peoples' lives in a myriad of ways, from providing easy access to services such as on-line banking and bookstores, or information such as product literature or drug and medical recommendations, to making a down payment on the promise of the "paperless office".

A side-effect of this dizzying rate of innovation and adoption is the fact that the Internet is continually out-growing its hardware and software infrastructure. The last two years have seen the proposal, development, and deployment of a vast number of new technologies, some of which are now considered mundane elements of the infrastructure, such as proxy servers. And the unprecedented and exponentially increasing quantities of information available through the Web have given new urgency to previously arcane topics, such as information retrieval or document classification.

This symposium addresses these problems from several angles: from understanding how existing systems are used (and abused), through developing new technologies and infrastructures to reduce load on the infrastructure, to developing new applications utilizing the new facilities.

This symposium would not have been possible without the dedication and effort of a huge cast, and I would like to thank all those who helped with this effort. The program committee went far beyond the call of duty, both reviewing papers and shepherding accepted papers. In addition to her position on the program committee, Margo Seltzer also provided valuable assistance as Board Liaison. Dan Klein put the tutorial track together. Of course the USENIX staff made everything happen: Judy DesHarnais handled all the conference logistics; Zanna Knight and Cynthia Deno organized and produced the Call for Papers, programs, and mailings; Eileen Cohen put the proceedings together; and Ellie Young served as coach, shortstop, and goalie. Thank you all for your herculean efforts!

Carl Staelin
Program Chair

Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web

Balachander Krishnamurthy

AT&T Labs–Research
180 Park Ave
Florham Park, NJ 07932 USA
bala@research.att.com

Craig E. Wills

Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609 USA
cew@cs.wpi.edu

Abstract

This paper presents work on piggyback cache validation (PCV), which addresses the problem of maintaining cache coherency for proxy caches. The novel aspect of our approach is to capitalize on requests sent from the proxy cache to the server to improve coherency. In the simplest case, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached, but potentially stale, resources from that server for validation.

Trace-driven simulation of this mechanism on two large, independent data sets shows that PCV both provides stronger cache coherency and reduces the request traffic in comparison to the time-to-live (TTL) based techniques currently used. Specifically, in comparison to the best TTL-based policy, the best PCV-based policy reduces the number of request messages from a proxy cache to a server by 16-17% and the average cost (considering response latency, request messages and bandwidth) by 6-8%. Moreover, the best PCV policy reduces the staleness ratio by 57-65% in comparison to the best TTL-based policy. Additionally, the PCV policies can easily be implemented within the HTTP 1.1 protocol.

1 Introduction

A proxy cache acts as an intermediary between potentially hundreds of clients and remote web servers by funneling requests from clients to various servers. In the process, the proxy caches frequently requested resources to avoid contacting the server repeatedly for the same resource if it knows, or heuristically decides, that the information on the page has not changed on the server.

A problem with caching resources at a proxy or within a browser cache, is the issue of cache coherency—how does the proxy know that the cached resource is still current [6]? If the server knows how long a resource is valid (e.g., a newspaper generated at 5:00am daily), the server can provide a precise expiration time. Cached copies are always fresh until the expiration time. More commonly, the resource that is made available has no clear expiration time. It may change in five minutes or remain unchanged for a long time.

Our work addresses the problem of maintaining cache coherency for proxy caches. The novel aspect of our approach is using requests sent from the proxy cache to the server to obtain additional coherency information. In the simplest approach, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached, but potentially stale, resources from that server for which the expiration time is unknown. Compared to other techniques, this piggyback cache validation (PCV) approach has the potential of both ensuring stronger cache coherency and reducing costs.

The paper is organized as follows: Section 2 discusses related work in the area of cache coherency in the Web context. Section 3 describes piggyback cache validation and presents two possible implementations on top of the Hypertext Transport Protocol (HTTP). Section 4 presents a study of two variants of PCV and contrasts them with four other cache coherency approaches. The study is based on trace-driven simulation of two large logs using evaluation criteria discussed in Section 5. Results from these simulations are presented in Section 6. Section 7 summarizes the work with a discussion of ongoing work and future directions.

2 Related Work

Caching and the problem of cache coherency on the World Wide Web are similar to the problems of caching in distributed file systems [10, 18]. However, as pointed out in [8], the Web is different than a distributed file system in its access patterns, its larger scale, and its single point of updates for Web objects.

Cache coherency schemes providing two types of consistency have been proposed and investigated for caches on the World Wide Web. One type—strong cache consistency, is maintained via one of two approaches. In the first approach, *client validation*, the proxy treats cached resources as potentially out-of-date on each access and sends a If-Modified-Since header with each access of the resource. This approach provides strong cache consistency, but can lead to many 304 responses (HTTP response code for "Not Modified") by the server if the resource does not actually change. The second approach is *server invalidation*, where upon detecting a resource change, the server sends invalidation messages to all clients that have recently accessed and potentially cached the resource [14]. This approach requires a server to keep track of lists of clients to use for invalidating cached copies of changed resources and can become unwieldy for a server when the number of clients is large. In addition, the lists can become out-of-date causing the server to send invalidation messages to clients who are no longer caching the resource.

In contrast, weak consistency approaches seek to minimize the proxy validation and server invalidation messages by using a heuristic or a pre-defined value as an artificial expiration time on a cached resource. One such approach, based on the last modified time [8], is for the proxy to adopt an adaptive time-to-live (TTL) expiration time (also called the Alex protocol [4]). The older the resource, the longer the time period between validations. This adaptive TTL heuristic is reasonable, but there can be periods where the cached resource is potentially stale. Related work in the WebExpress project for mobile environments allows users to set a fixed coherency interval for objects with the capability to change this interval for specific objects [9].

In terms of prior work using piggybacking to improve cache coherency, Mogul [15] has proposed piggybacking server invalidations for cached resources as part of replies to client requests. This idea was a motivation for our work on piggyback cache validation, but not directly investigated in the results reported here.

The concept of piggybacking additional information to a Web request or reply has been proposed in limited forms for other uses. Previous work suggests that server knowledge about access patterns for a requested resource could be returned along with the resource. This knowledge could be used by the client to control prefetching [2, 19]. [22] extends this approach by using both client and server knowledge. Mogul proposes "hit-metering" as a technique for a cache to report reference count information back to an origin server [16]. This information can be used in predicting reference hit information, which can be passed as hints to a cache whenever the server sends a response to a cache. Finally, the WebExpress [9] work proposes a batch approach to perform a single check at the beginning of a session for all cached objects older than the coherency interval. This batching is related to our idea of a validation list and could be carried out using a piggybacked approach.

3 Piggyback Cache Validation

Our approach for maintaining cache coherency while reducing the number of messages exchanged with a server is to piggyback cache state information onto HTTP requests to the server [12]. While individual browser clients could use our approach, it is most beneficial to proxy caches because the number of resources cached from a particular server is small and likely short-lived in an individual browser cache. Thus, we focus on the use of the piggyback mechanism for a proxy cache.

In the simplest approach, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached resources from that server, for which the expiration time is unknown and the heuristically-determined TTL has expired. The server handles the request and indicates which cached resources on the list are now stale allowing the proxy to update its cache.

The proxy treats client requests for cached resources with a validated age of less than the time to live threshold as current. Requests for cached resources that have not recently been validated cause an If-Modified-Since (IMS) Get request to be sent to the server.

The performance of piggyback cache validation depends on the number of resources cached at a proxy for a particular server and the number of requests that are sent by the proxy to the server. If there are few such requests, meaning the cache contents do not get validated with piggybacking, then the approach performs similar to TTL-based policies in generating a validation check when the time-to-live expires. However, if there is traffic between the proxy and server, then the cache contents are validated at the granularity of the expiration time threshold without need for IMS requests. Results from prior studies indicate such traffic exists and best case results have yielded a 30-50% proxy cache hit rate [1]. We have found a similar hit rate in our studies. The introduction of piggyback validation allows relatively short expiration times to be used resulting in close to strong cache coherency while reducing the number of IMS requests sent to a server in comparison to existing TTL-based policies.

The added cost of our mechanism is mainly in the increased size of the regular request messages due to piggybacking. However, there are no new connections made between proxies and servers. The number of piggybacked validations appended to a single request can be controlled by the proxy cache. The cost for the proxy cache is also slightly increased as it must maintain a list of cached resources on a per server basis. The additional cost for the server is that it must validate the piggybacked resources in addition to processing the regular request. However, in the absence of piggybacking, such validations may have to be done in the future by the server in separate connections.

Implementation of piggyback cache validation can be done independent of a particular cache replacement policy [3, 21]. In our initial work we have used a standard LRU cache replacement policy. However, validation information provided by a server could be used by such a replacement policy. For example, if a proxy cache finds that a cached resource is frequently invalidated then this resource would be a good candidate for cache replacement.

Two approaches could be used to implement the PCV mechanism within HTTP. The first approach is to implement the mechanism via a new HTTP header type for validation list requests and replies. In a request, the header field consists of a list of resource and last modified time pairs. On a reply, the field would contain a list of invalid resources or a value indicating that all resources in the request list are valid. This approach is compact, but it could require the server to validate the entire piggybacked list before it replies to the request itself. Alternately, invalid resources could be return as part of a footer as allowed in HTTP/1.1 [11].

Another approach is to pipeline HEAD requests trailing the resource request. The approach requires more bandwidth, but it can be implemented in HTTP 1.1 with no changes to the protocol. It also separates the request from cache validation.

In either implementation, if a server does not implement the mechanism, the proxy cache works fine, albeit without the piggybacked validation information. In our testing we assume the first approach.

4 Testing

4.1 Proxy Cache Logs

We constructed a trace-driven simulation to test our ideas and used two sets of logs:

- Digital Equipment Corporation proxy logs [5] (Sept. 16–22, 1996) with 6.4 million Get requests for an average rate of 40031 requests/hour and 387.0 MByte/hour. 57832 distinct servers were contacted with the top 1% of the servers being responsible for over 59% of the resources accessed. 45% of the servers had fewer than 10 resources accessed and over 89% of the servers had fewer than 100 resources accessed. 3.4% of the servers (1943) accounted for over half the 2083491 unique resources accessed.

- AT&T Labs–Research packet level trace [17] (Nov. 8–25, 1996) with 1.1 million Get requests for an average rate of 2805.63 requests/hour and 18.4 MByte/hour. 18005 distinct servers were contacted with the top 1% of the servers being responsible for over 55% of the resources accessed. 48% of the servers had fewer than 10 resources accessed and over 92% of the servers had fewer than 100 resources accessed. 5.6% of the servers (1019) accounted for over half the 521330 unique resources accessed.

4.2 Cache Coherency Policies

We tested six cache coherency policies with these logs:

1. pcvfix—This policy implements the basic piggyback cache idea with a fixed TTL (time to live) expiration period. By default, a cached resource is considered stale once a period of one hour has elapsed. When the expiration time is reached for this resource, a validation check is piggybacked on a subsequent request to its server. If the resource is accessed after its expiration, but before validation, then a If-Modified-Since Get request is sent to the server for this resource.

2. pcvadapt—This policy implements the basic piggyback cache idea, but with an adaptive TTL expiration time based on a fraction (adaptive threshold) of the age of the resource. As in the Alex FTP protocol [4, 8], the motivation is that newer resources change more frequently, while older resources change less often. However, because we believe piggybacked validations are relatively inexpensive, the pcvadapt policy uses a maximum expiration time equal to the fixed TTL in effect. This policy allows a relatively tight limit for all resources with an even shorter expiration for newer resources.

3. ttlfix—This policy uses a fixed TTL expiration period for all resources and does no piggybacking. If a cached resource is accessed after its expiration then a If-Modified-Since Get request is sent to the server for this resource.

4. ttladapt—This policy uses an adaptive TTL expiration time based on resource age with no piggybacking. The upper bound for a resource expiration time is fixed at one day. The pcvadapt policy has a tighter bound because piggyback validation checks are less expensive than If-Modified-Since Get requests, which are the only means of validation for the ttladapt approach. The ttladapt policy is used in the Squid Internet Object Cache [20], which allows the adaptive threshold and maximum age to be configurable.

5. alwaysvalidate—This policy generates a If-Modified-Since Get request for every access of a cached resource. The policy ensures strong coherency, but causes a request to the server for every resource access. It is used to measure other policies against.

6. nevervalidate—This policy has an infinite expiration time so that cached resources are never validated. The policy minimizes costs by always using the cached copy of a resource, but results in the most use of stale copies. It is another policy against which other policies are measured.

4.3 Parameters Used

The cache coherency policies were studied by varying four parameters: cache size, PCV size (the maximum size of a single piggyback list), the TTL value, and the adaptive threshold. To focus the presentation of results while still being able to vary all parameters, we established a base set of parameters from which we varied one parameter at a time. The base values were established based on other published work and after testing the policies under different conditions. The base parameter values (and the range studied) were:

- cache size of 1GB (range 1MB to 8GB),
- maximum PCV size of 50 (range 10 to 1000),
- TTL of one hour (range 0.5 to 24 hours), and
- adaptive threshold of 0.1 (range 0.05 to 1.0).

We did not vary the cache replacement policy for this study, but used LRU for all our tests. Variation of replacement policy and its interaction with the cache coherency policies is an area for future work.

5 Evaluation Criteria

There are three types of costs traditionally considered when evaluating the performance of Web resource retrieval:

- response latency—how long it takes to retrieve the requested resource,
- bandwidth—how many bytes must be served by the server and transmitted over the network, and
- requests—how many requests must be handled by the server and transmitted over the network.

In this context, the goal of a good cache coherency policy, when combined with a cache replacement policy, is to provide up-to-date (non-stale) resources to clients while minimizing the above costs. However, translating the simulation results for a policy on a set of data to their relative "goodness" can be done in many ways. In the following, we define and justify how we determine the cost, staleness and overall goodness metrics used in comparing different cache coherency policies.

5.1 Cost Evaluation

The typical measure for cache replacement policies is the "hit rate," or the percentage of time that a resource request can be provided from cache. Because this measure does not account for resource size, the use of byte hit rate is also common in recent literature [3, 21]. These measures can be used to derive cost savings for bandwidth and requests, but do not reflect on savings in response latency. Mogul also points out that measuring cache hits does not measure the effect of caching mechanisms on the cost of a cache miss [16].

In addition, the cache replacement policy studies have ignored the cost for cache coherency and ratio of stale resources. These studies do not distinguish between cache hits for resources that can be used directly and hits for resources that are validated with the server. Previous studies on cache coherency [8, 14] report statistics on stale cache hits along with information on network, server, and invalidation costs. However, these studies do not report these values in the context of other cache activity, specifically cache replacement.

Our approach is to use a comprehensive cost model that accounts for the combined costs of cache replacement and coherency. The model incorporates the costs for the three possible actions that can occur when a resource is requested by a client from a proxy cache:

1. Serve from cache—the resource is currently cached and returned to the client without contacting the server. We define the costs of such action to be zero.

2. Validate—the resource is currently cached and is returned to the client after the proxy validates the cached copy is current by contacting the server. This action involves a request to the server, along with a latency cost corresponding to the distance to the server and a small bandwidth cost.

3. Get—the resource is not in cache (or the cached copy is invalid). The resource is returned to the client after retrieval from the server. This action involves a request to the server, along with bandwidth and transfer latency costs corresponding to the size of the resource and the distance from the server.

Considering these three actions, we use a normalized cost model for each of the three evaluation criteria and each of the actions where $c[a, e]$ represents the cost for action a and evaluation criterion e. We let C denote the matrix representing all combinations of proxy cache actions and evaluation criteria. In our work, each $c[a, e]$ is computed using the data from our test logs. These costs are shown in Tables 1 and 2 and explained below.

For each evaluation criterion, the cost of an average Get request with full resource reply (status 200) is normalized to 1.00 based on the values in the log. In the Digital logs, the actual values are 12279 bytes of bandwidth (includes contents and headers), 3.5 seconds of latency and one request for an average retrieval. In the AT&T logs, the actual values are 8822 bytes of bandwidth, 2.5 seconds of latency and one request.

Table 1: Normalized Cost Matrix C for Digital Logs

	Evaluation Criterion (e)			
Action (a)	Re-sponse	Band-width	Re-quest	Avg.
Get	1.00	1.00	1.00	1.00
Validate	0.36	0.03	1.00	0.47
In Cache	0.00	0.00	0.00	0.00

Table 2: Normalized Cost Matrix C for AT&T Logs

	Evaluation Criterion (e)			
Action (a)	Re-sponse	Band-width	Re-quest	Avg.
Get	1.00	1.00	1.00	1.00
Validate	0.12	0.04	1.00	0.39
In Cache	0.00	0.00	0.00	0.00

The cost of using a resource from cache is defined to be zero. This definition focuses on the external costs for resource access, although internal networks, computers and even the proxy cache itself contribute some latency [13].

The intermediate cost of a Validate request, which returns a validation of the current cache copy (response 304), is computed relative to the cost of a full Get request. As shown in Tables 1 and 2, this action is just as expensive as a full Get request in terms of requests, of intermediate cost in terms of response latency and of little cost in terms of bandwidth.

Tables 1 and 2 show a fourth evaluation criterion, which is the average of the costs for the other three criterion. This criterion is introduced as a composite criterion that assigns equal importance to each of the standard criteria.

Given the matrix C, we can compute the total cost for a cache coherency policy p by knowing the relative proportion of each cache action a for the policy. We let $w[p, a]$ represent the proportional weight for the occurrence of action a while using policy p and let W denote the matrix of all combinations of policies and actions. The matrix W varies depending on the simulation parameters used, but for illustration, Tables 3 and 4 show W when using the base set of parameters discussed in Section 4.3.

Using the matrices C and W, the total cost $t[p, e]$

Table 3: Representative Weight Matrix W for Digital Logs

	Action (a)		
Policy (p)	Get	Validate	In Cache
pcvfix	0.5407	0.0070	0.4522
pcvadapt	0.5429	0.0079	0.4492
ttlfix	0.5457	0.0953	0.3590
ttladapt	0.5520	0.0296	0.4184
alwaysvalidate	0.5542	0.4458	0.0000
nevervalidate	0.5391	0.0000	0.4609

Table 4: Representative Weight Matrix W for AT&T Logs

	Action (a)		
Policy (p)	Get	Validate	In Cache
pcvfix	0.5183	0.0337	0.4480
pcvadapt	0.5221	0.0351	0.4428
ttlfix	0.5238	0.2188	0.2574
ttladapt	0.5279	0.1182	0.3539
alwaysvalidate	0.5308	0.4692	0.0000
nevervalidate	0.4980	0.0000	0.5020

for each policy p and evaluation criterion e is easily computed with the matrix product $T = W \cdot C$. The resulting $t[p, e]$ values are used in reporting costs for cache coherency policies in this paper.

In analyzing Tables 3 and 4, the ratio of resources causing a full Get request is relatively constant for all cache coherence policies across the two logs. This figure is primarily dependent on the performance of the cache replacement policy. As defined, the alwaysvalidate policy never serves a resource directly from the cache, while the nevervalidate policy never generates a Validate request. Of more interest to our overall performance study, the PCV policies generate the least number of Validate requests.

In calculating the performance results reported in this paper two adjustments are made. First, the cost of a full Get request represents the costs for actual resources retrieved. Thus, if the average size or latency for resources actually retrieved is larger than the average for the entire log then $c[\text{Get}, \text{Bandwidth}]$ and $c[\text{Get}, \text{Response}]$ are greater than one. This situation may occur if a cache replacement policy caches smaller resources to increase the hit rate, but results in higher costs for cache misses.

Second, it is not fair to measure the impact of PCV policies without accounting for their increased costs. Consequently the bandwidth and response costs are increased for these policies based on the size in bytes of a piggybacking validation, the response time for the server to do the validation and the number of piggyback validations per request. In the simulation, 50 bytes are added to the request packet and 0.1ms is added to the response time for each piggybacked validation. The number of validations varies, but for the pcvadapt policy using the standard parameters, the average number of piggybacked validations requests is 1.1 for the Digital and 7.0 for the AT&T logs. The difference between the two averages seems to stem from the fact that there is a higher degree of locality of reference in the AT&T logs than in the Digital logs, possibly due to the higher user population in the latter case.

5.2 Staleness Evaluation

The staleness ratio is the number of known stale cache hits divided by the number of total requests (both serviced from cache and retrieved from the server). We chose not to measure staleness as the more traditional ratio of stale cache hits divided by number of cache hits because of differences in the in-cache hit ratio (shown in Tables 3 and 4) caused by differences in the validation and invalidation approaches of the policies. Using our staleness ratio definition allows for a fairer comparison of the policies, although it deflates the ratios in comparison to measuring the ratio of stale cache hits.

5.3 Goodness Evaluation

We believe that the cost and staleness evaluations provide fair and appropriate measures to compare the various policies. However, a good cache coherency policy should minimize both cost (relative to the cache replacement policy) and staleness. To combine cost and staleness into a single metric, we compute an overall "goodness" metric, which combines the average cost and staleness relative to the range defined by the alwaysvalidate and nevervalidate policies for a given cache size. These two policies define the minimum and maximum costs and staleness relative to a cache replacement policy and cache size. It is subjective as to whether minimizing cost or staleness is more important. In the results presented in this paper they are given equal weight when computing the goodness metric.

6 Results

6.1 Variation of Cache Size Parameter

The following results are shown for variation in cache size from 1MB to 8GB. Figures 1–4 show the response time, bandwidth, request message and average costs for the respective policies using the Digital logs. As expected, the alwaysvalidate policy performs the worst while nevervalidate performs the best. In fact, Figure 3 shows that the alwaysvalidate policy provides the worst possible performance (cost of 1.0) for the request message cost because it generates a request (either Get or Validate) for each requested resource.

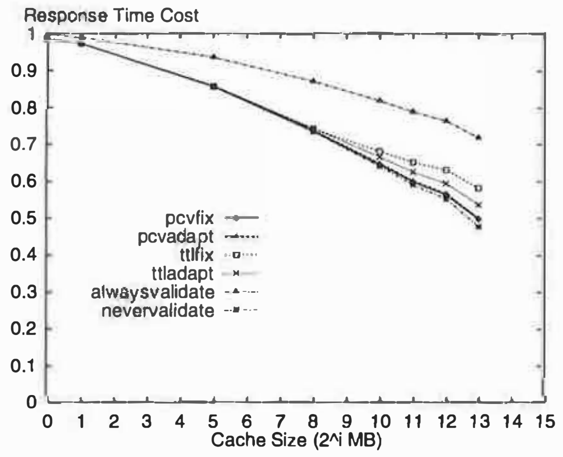


Figure 1: Response Time Cost versus Cache Size for Digital Logs

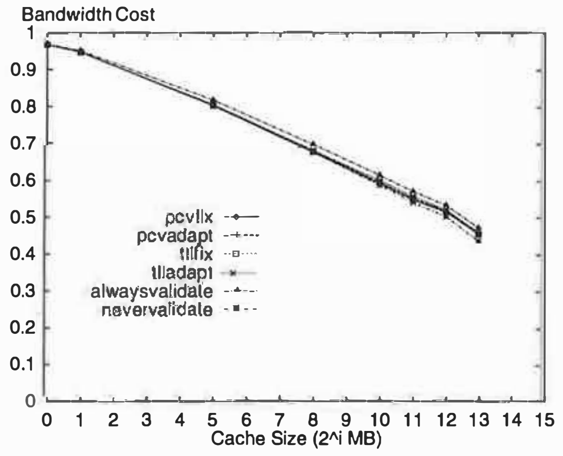


Figure 2: Bandwidth Cost versus Cache Size for Digital Logs

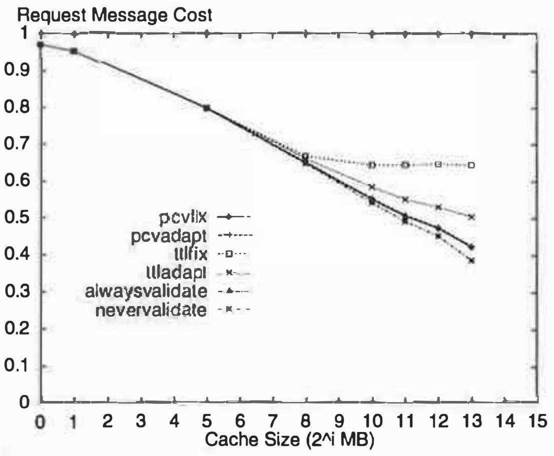


Figure 3: Request Message Cost versus Cache Size for Digital Logs

The differences between policies in the figures is primarily a function of ratio of Validate requests that each generates and the evaluation criterion cost for such a request. Thus, there is little distinction between the policies in Figure 2 because the bandwidth costs for a Validate request are minimal. On the other hand, there is more differentiation between the policies for the other evaluation criteria because the Validate request costs for these criteria are nontrivial.

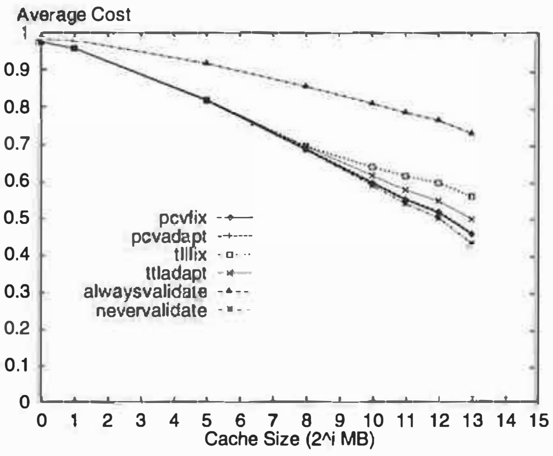


Figure 4: Average Cost versus Cache Size for Digital Logs

In comparing the policies, the PCV policies are next to the nevervalidate policy in incurring the least cost. This result is because the PCV policies reduce the number of Validate requests (also see Table 3) in comparison to the TTL-only policies therefore making better use of the cache contents. Specifically, the pcvadapt policy reduces the number of request messages by 16% and the average cost by 8% in comparison to the ttladapt policy for a 8GB cache.

In reporting the results, we note that the costs for the PCV policies are overstated. Due to the nature of the simulation we do not discover the invalidation of a cached resource until the next time that resource is accessed. If a cached resource is invalidated and then removed by the cache replacement policy before its next access then the invalidation is not discovered at all in our simulation. The cost results are pessimistic because the cache space for these invalidated resources is not freed at the time of invalidation as would be the case in an actual implementation. With this limitation, we measure an invalidation rate of 0.26 invalidations for every 100 resources accessed (not just those in cache) in the Digital logs for the standard parameter set and the pcvadapt policy.

Figure 5 shows the ratio of all requested resources returned as stale from the cache relative to the cache size. While nevervalidate has low staleness values for small caches, it rises to 4% for a cache size of 8GB in the Digital logs. The alwaysvalidate policy results in no stale resources. The other policies all have relatively low staleness ratios (less than 1%) with the pcvadapt policy clearly providing the strongest coherency. In comparison to the ttladapt policy, the pcvadapt policy reduces the staleness ratio by 65% for a 8GB cache.

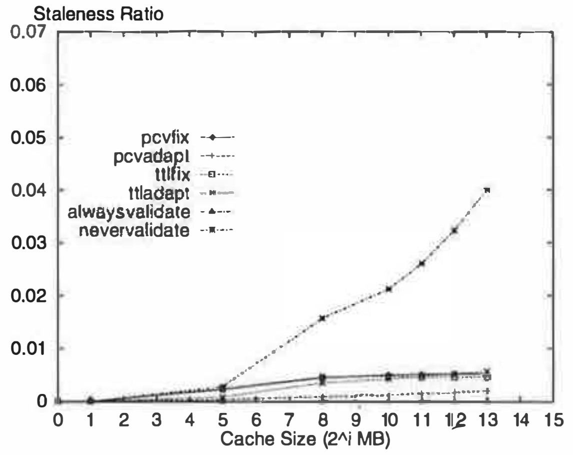


Figure 5: Staleness Ratio versus Cache Size for Digital Logs

Figure 6 shows the goodness metric of combining average cost and staleness performance measures using a weighted sum with each performance metric having equal weight. By definition, the alwaysvalidate and nevervalidate policies have a value of 0.5 at all cache sizes because they define the upper and lower bounds for goodness. As shown, the pcvadapt policy exhibits a high degree of goodness for large cache sizes indicating it is an excellent policy in providing up-to-date resources at a low cost. The pcvfix policy is the second best. For larger cache sizes the nevervalidate staleness ratio goes up (Figure 5), which causes less differentiation in the relative policy staleness ratios. This effect causes the relative differences in policy costs to be reflected more in the goodness metric and contributes to the dropoff in the goodness metric for TTL-based policies.

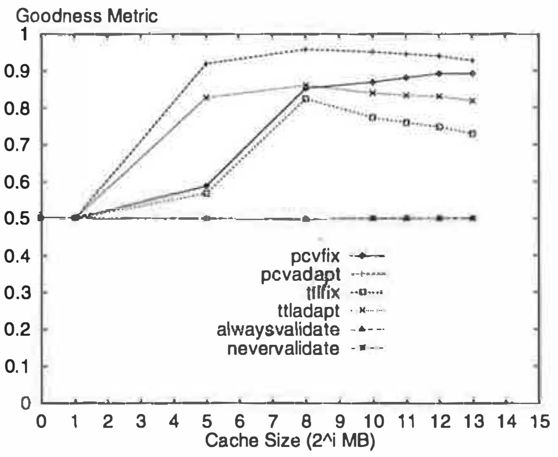


Figure 6: Goodness Metric versus Cache Size for Digital Logs

Figures 7–10 show the response time, bandwidth, request message and average costs for the respective policies as the cache size is varied using the AT&T logs. The relative ordering of the policies is the same as for the Digital logs with the differences between the policies generally more pronounced. This differentiation is a result of relatively more Validate requests being generated for the AT&T data as shown in Table 4. In comparison with the Digital data, the bandwidth costs can be differentiated in Figure 8 with the PCV policies resulting in the highest bandwidth costs. This result indicates that the piggybacked validations for the AT&T data add enough bytes to slightly increase the costs relative to the other policies, although the PCV policies provide lowest response time, request and average costs. Specific comparisons for a 8GB cache show the pcvadapt policy reduces message cost by 17% and the average cost by 6% in comparison to the ttladapt policy. The invalidation rate for the AT&T logs is 2.4 per 100 accesses, which is higher than for the Digital logs due to more piggyback validations being generated.

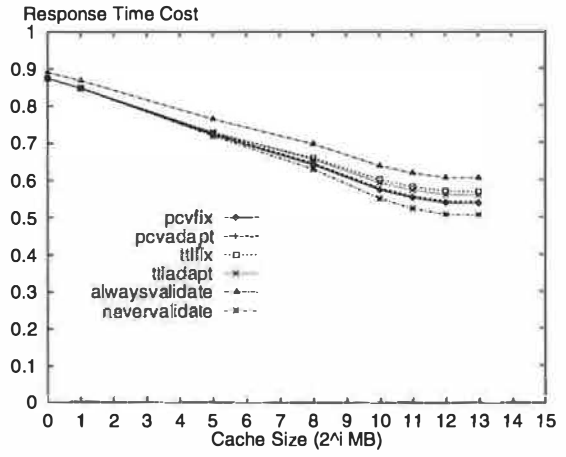


Figure 7: Response Time Cost versus Cache Size for AT&T Logs

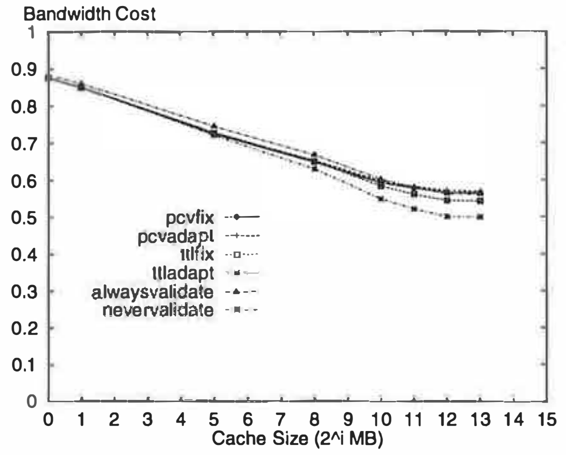


Figure 8: Bandwidth Cost versus Cache Size for AT&T Logs

Figure 11 shows similar staleness results for the AT&T logs as for the Digital logs with the nevervalidate policy returning nearly 7% stale resources for large cache sizes. The other policies return close to 1% stale resources with the pcvadapt policy providing the most strongly coherent results (57% improvement over the ttladapt policy for a 8GB cache) next to the alwaysvalidate policy. The resulting goodness metric results for the AT&T logs are shown in Figure 12 with comparable results to those shown for the Digital logs.

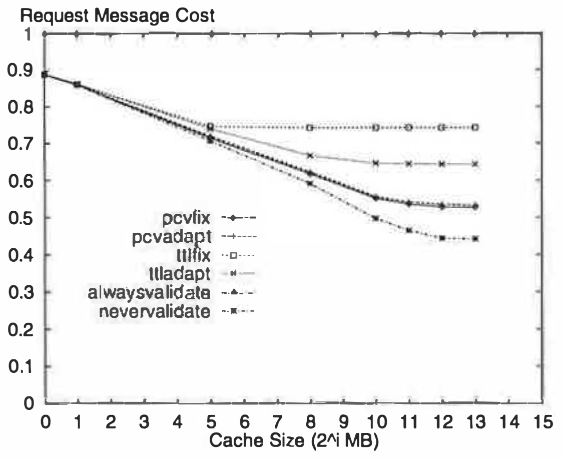


Figure 9: Request Message Cost versus Cache Size for AT&T Logs

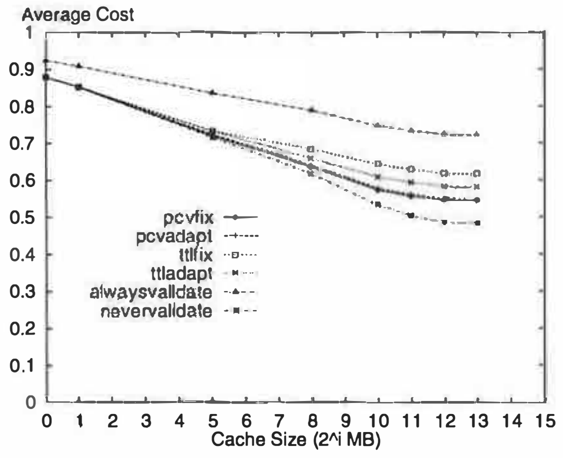


Figure 10: Average Cost versus Cache Size for AT&T Logs

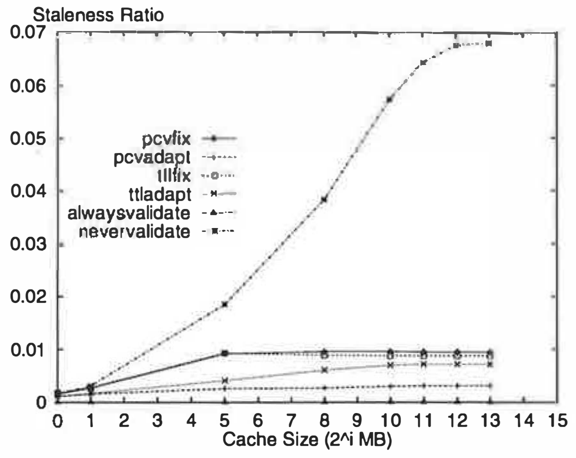


Figure 11: Staleness Ratio versus Cache Size for AT&T Logs

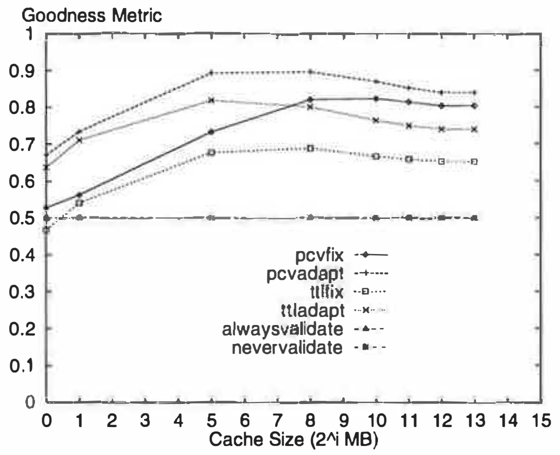


Figure 12: Goodness Metric versus Cache Size for AT&T Logs

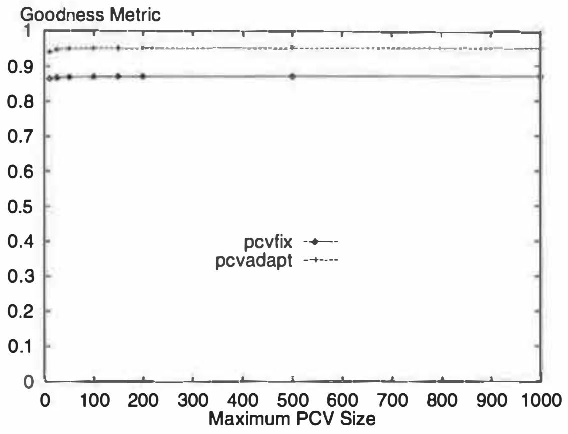


Figure 13: Goodness Metric versus Maximum PCV Size for Digital Logs

6.2 Variation of Other Parameters

The remaining results summarize the effects on the goodness metric of varying the maximum PCV list size, time-to-live and adaptive threshold parameters for the Digital and AT&T logs (the specific cost and staleness results are not shown). These results reflect the output of simulations where the dependent parameter was varied while the other parameters were set to their base values where the base cache size is 1GB. The goodness metric is again calculated relative to the alwaysvalidate and nevervalidate policies for a cache of this size.

Figures 13 and 14 show there is little variation in results based on the maximum PCV list size. Smaller maximums than the default of 50 result in slightly degraded performance, but reflect that there is generally enough traffic between the client proxy and the server to satisfy the piggybacking needs of the proxy. Allowing larger PCV lists does not improve the performance of the PCV policies.

Figures 15 and 16 show some variation between the policies for smaller TTL parameters, but the performance ordering of the policies remains the same. As the TTL value is raised, the two adaptive policies perform similarly as do the two fixed TTL approaches. Use of a bigger TTL value reduces the potential benefit of the PCV policies.

Finally, Figures 17 and 18 show the expected that the lower the adaptive threshold the better the relative results for the adaptive policies. The base value of 0.1 is relatively good in the range.

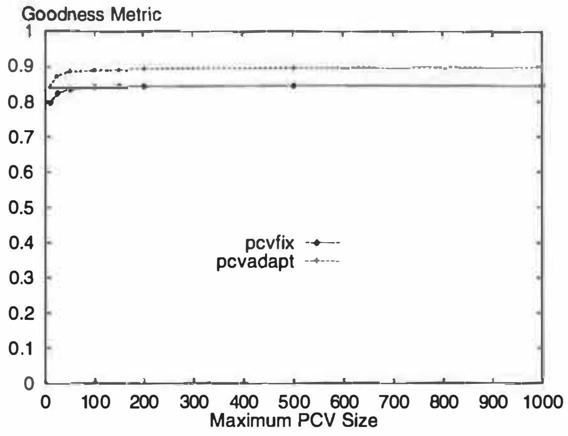


Figure 14: Goodness Metric versus Maximum PCV Size for AT&T Logs

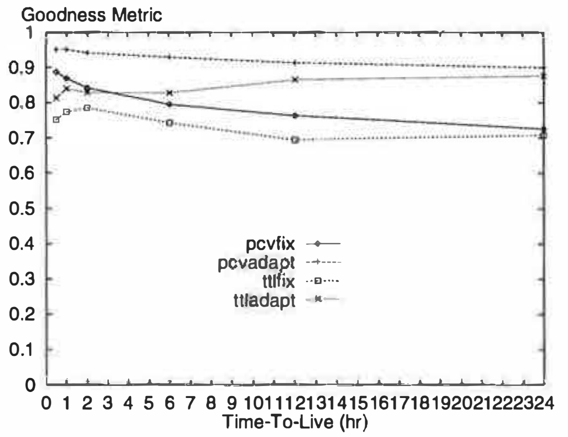


Figure 15: Goodness Metric versus Time-To-Live for Digital Logs

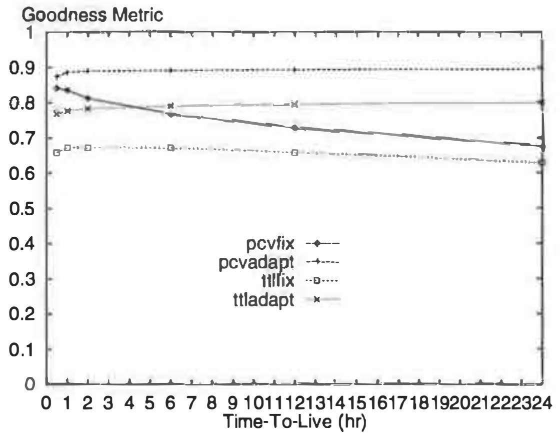


Figure 16: Goodness Metric versus Time-To-Live for AT&T Logs

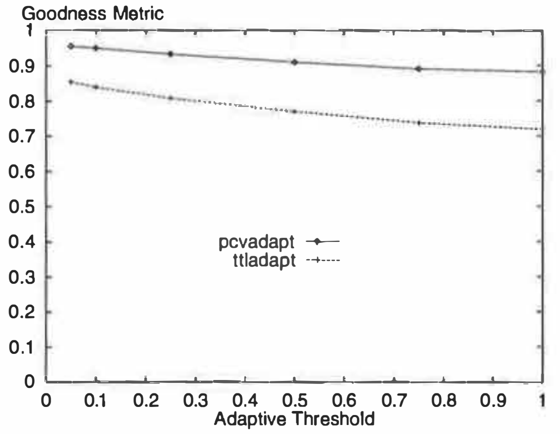


Figure 17: Goodness Metric versus Adaptive Threshold for Digital Logs

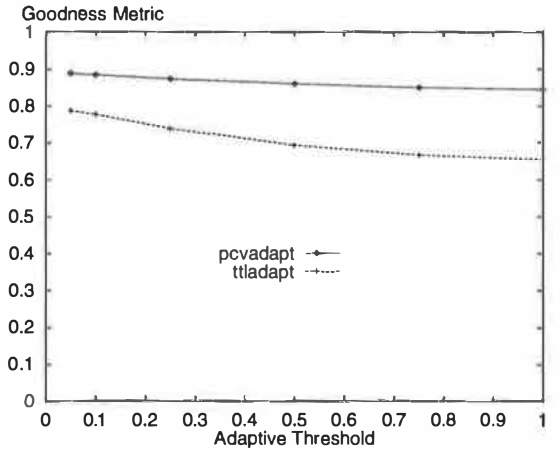


Figure 18: Goodness Metric versus Adaptive Threshold for AT&T Logs

7 Summary and Future Work

We believe the similarity of results on two large, independent data sets clearly demonstrates the merits of the piggyback cache validation approach. By combining piggybacking with an adaptive threshold, the pcvadapt policy is clearly the best at providing close to strong coherency at a relatively low cost when we consider response latency, request messages, bandwidth and average costs. In comparison to the ttladapt policy, the best TTL-based policy, the pcvadapt policy, the best PCV-based policy, reduces the number of messages to the server by 16-17% and the average cost by 6-8%. The pcvadapt policy reduces the staleness ratio by 57-65% in comparison to the ttladapt policy. Additionally, the PCV policies can easily be implemented with the HTTP 1.1 protocol.
Other directions for piggybacking of information include a server piggybacking resource invalidations on replies to a client. These invalidations could be for all the resources at its site or selected subsets of resources [15]. Clients then use the list of invalidations to remove stale copies. Piggybacking of such information onto existing server replies does not introduce new network traffic and alleviates servers having to maintain client lists in comparison to previous server invalidation work [14].
Although not studied in this work, cache coherency is related to cache replacement as a proxy cache works to provide up-to-date resources at the lowest cost. Piggybacking has the potential to positively affect cache replacement decisions. For example, frequently invalidated resources would be good candidates for cache replacement. Proxy caches can also use information about resource usage piggybacked on server replies in making replacement decisions. Additionally, recent work [7] shows that resources that change are accessed more often, there is variation in the rate of change across content types, and the most frequently referenced resources cluster around specific periods of time—all of which can be used while deciding what and when to piggyback.

8 Acknowledgments

We thank Digital Equipment Corporation for making their proxy traces available to us. We thank Anja Feldmann for help with the AT&T trace, Steve Bellovin and Phong Vo for discussions, Jeffrey Mogul, Misha Rabinovich, Jennifer Rexford, Fred Douglis, and the anonymous reviewers for making comments on draft versions of the paper.

References

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the Fourth International World Wide Web Conference*, December 1995. `http://www.w3.org/pub/Conferences/WWW4/Papers/155/`.

[2] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of the 4th ACM International Conference on Information and Knowledge Management*, November 1995.

[3] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Symposium on Internet Technology and Systems.* USENIX Association, December 1997.

[4] V. Cate. Alex – A global filesystem. In *Proceedings of the USENIX File System Workshop*, pages 1–12. USENIX Association, May 1992.

[5] Digital Equipment Corporation. Proxy cache log traces, September 1996. `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`.

[6] Adam Dingle and Tomas Partl. Web cache coherence. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996. `http://www5conf.inria.fr/fich_html/papers/P2/Overview.html`.

[7] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeff Mogul. Rate of change and other metrics: a live study of the world wide web. In *Symposium on Internet Technology and Systems.* USENIX Association, December 1997.

[8] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the USENIX Technical Conference*, pages 141–152. USENIX Association, January 1996. `http://www.usenix.org/publications/library/proceedings/sd96/seltzer.html`.

[9] Barron C. Housel and David B. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the ACM/IEEE MOBICOM '96 Conference*, October 1996. `http://www.networking.ibm.com/art/artwewp.htm`.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):55–81, February 1988.

[11] Internet Engineering Task Force. Hypertext transport protocol – HTTP/1.1, January 1997. `http://ds.internic.net/internet-drafts/`.

[12] Balachander Krishnamurthy and Craig E. Wills. Piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the 2nd Web Caching Workshop*, Boulder, CO, June 1997. National Laboratory for Applied Network Research. `http://ircache.nlanr.net/Cache/Workshop97/Papers/Wills/wills.html`.

[13] Thomas M. Kroeger, Darrel D.E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Symposium on Internet Technology and Systems.* USENIX Association, December 1997.

[14] Chengjie Liu and Pei Cao. Maintaining strong cache consistency in the world-wide web. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[15] Jeffrey Mogul. An alternative to explicit revocation?, January 1996. `http://weeble.lut.ac.uk/lists/http-caching/0045.html`.

[16] Jeffrey C. Mogul. Hinted caching in the web. In *Proceedings of the 1996 SIGOPS European Workshop*, 1996. `http://mosquitonet.stanford.edu/sigops96/papers/mogul.ps`.

[17] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *ACM SIGCOMM'97 Conference*, September 1997. `http://www.acm.org/sigcomm/sigcomm97/papers/p156.html`.

[18] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[19] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *Computer Communication Review*, 26(3):22–36, 1996.

[20] Squid internet object cache. `http://squid.nlanr.net/Squid`.

[21] Stephen Williams, Marc Abrams, Charles R. Standbridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of the ACM SIGCOMM Conference*, pages 293–305, August 1996. `http://www.acm.org/sigcomm/sigcomm96/williams/p156.html`.

[22] Craig E. Wills and Joel Sommers. Prefetching on the web through merger of client and server profiles, June 1997. `http://www.cs.wpi.edu/~cew/papers/webprofile.ps`.

Exploring the Bounds of Web Latency Reduction from Caching and Prefetching

Thomas M. Kroeger[†]
Department of Computer Engineering
University of California, Santa Cruz

Darrell D. E. Long[‡]
Department of Computer Science
University of California, Santa Cruz

Jeffrey C. Mogul[§]
Digital Equipment Corporation
Western Research Laboratory

Abstract

Prefetching and caching are techniques commonly used in I/O systems to reduce latency. Many researchers have advocated the use of caching and prefetching to reduce latency in the Web. We derive several bounds on the performance improvements seen from these techniques, and then use traces of Web proxy activity taken at Digital Equipment Corporation to quantify these bounds.

We found that for these traces, local proxy caching could reduce latency by at best 26%, prefetching could reduce latency by at best 57%, and a combined caching and prefetching proxy could provide at best a 60% latency reduction. Furthermore, we found that how far in advance a prefetching algorithm was able to prefetch an object was a significant factor in its ability to reduce latency. We note that the latency reduction from caching is significantly limited by the rapid changes of objects in the Web. We conclude that for the workload studied caching offers moderate assistance in reducing latency. Prefetching can offer more than twice the improvement of caching but is still limited in its ability to reduce latency.

1 Introduction

The growth of the Web over the past few years has inspired researchers to investigate prefetching and caching as techniques to reduce latency [1, 12, 15]. While such techniques have seen significant success reducing latency in storage systems [7, 8, 9, 14] and in processor memory hierarchies [13], it remains to be seen how effective such techniques can be within the World Wide Web.

We classify caching and prefetching into four different methods and then derive bounds on these methods. Using traces taken over a three week period at Digital Equipment Corporation, we quantify these bounds.

We assume the use of a proxy server as an intermediary between the client (browser) and the web server. This proxy server accepts requests from clients and satisfies them using data that has been prefetched, cached or retrieved directly from an appropriate web server. This configuration is quite common in the Web today. Whether a proxy is used or not, this model serves to partition the latency of Web retrievals into two components: *external latency*, caused by network and web server latencies that are external to an organization, and *internal latency*, caused by networks and computers within the bounds of an organization. Figure 1 illustrates a common configuration.

Because the proxies are normally located on the organization's network, the communication between the client and proxy is normally a small portion of the overall latency. On the other side, the proxy-server communication normally accounts for a significant majority of the total event latency. The primary goal of proxy-based caching and prefetching is to reduce the amount of time the client waits for data by reducing or removing external latency. In our traces, external latency accounts for 77% of the latency seen in our entire trace set and 88% of the latency seen by subset of clients geographically close to the proxy.

With this potential for such a significant performance gain, the best improvement we saw from caching and prefetching reduced total latency by 60%. Additionally, we saw that the prefetching lead time, the amount of time between when prefetching begins and when the object is needed, significantly affects the amount of latency

[†] tmk@cse.ucsc.edu. Supported in part by Digital Equipment Corporation and the Office of Naval Research under Grant N00014–92–J–1807.

[‡] darrell@cse.ucsc.edu. Supported in part by the Office of Naval Research under Grant N00014–92–J–1807.

[§] mogul@pa.dec.com.

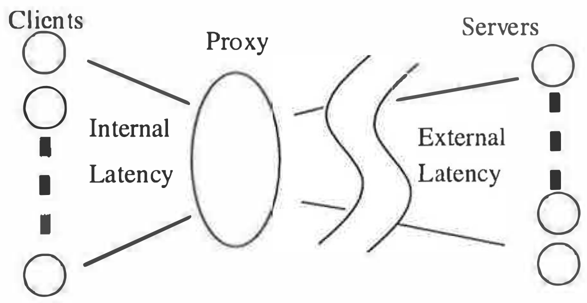


Figure 1: Typical proxy configuration

reduction. We found that when we limited our simulator to providing only a three-minute warning, the latency reduction dropped to 43%. Additionally, we observe that the latency reduction from caching was half of what it would have been for a data set with data objects that did not change. This observation agrees with several studies that show a high rate of change for objects in the web [11, 6, 3]. Comparing our results with the external latency we observe that, for the workload examined, Web latency consists of 23% internal latency, 20% external latency that cannot be cached or prefetched and 57% external latency that can be removed through prefetching and caching. The key point that we take from these results is that while caching and prefetching are helpful, under the current conditions there is a limit to their ability to reduce latency.

The rest of this article is organized as follows: section 2 categorizes caching and prefetching and presents four methods for using these techniques. We then present bounds for each method. In §3 we discuss the methodology used to quantify these bounds. In §4 we present the results of our simulations. Related work is then addressed in §5, and we present our conclusions in §6.

2 Bounding Caching and Prefetching

Our goal is to determine an upper bound on the effectiveness of proxy-based caching and prefetching for reducing latency in the Web. We classify caching and prefetching algorithms into different categories. We then construct models for bounding the performance of each category of algorithm. In order to ensure that our bounds are widely applicable, we do not present any specific algorithms and attempt to keep our analysis as general as possible.

2.1 Categories of Caching and Prefetching

We distinguish between caching algorithms based on whether a cache will remain *passive*, taking action only as a result of requests or if it is *active*, prefetching data in anticipation of future requests. We distinguish between prefetching algorithms based on where the information used to determine what to prefetch originates.

Traditionally, caching is thought of as a system that only reacts to requests. We define a *passive* cache as one that only loads a data object as a result of a client's request to access that object. In contrast, we use the term *active* cache to refer to caches that employ some mechanism to prefetch data. We note that passive caching systems also serve to reduce network traffic. However, for this work we focus on the use of passive caching for latency reduction.

We categorize prefetching into two categories, *local* and *server-hint*, based on where the information for determining which objects to prefetch is generated. In *local* prefetching, the agent doing the prefetching (*e.g.* a browser-client or a proxy) uses local information (*e.g.* reference patterns) to determine which objects to prefetch. Prefetching algorithms that do not make use of information from the server, whether employed at a client or at a proxy, would be considered local prefetching.

In *server-hint* based prefetching, the server is able to use its content specific knowledge of the objects requested, as well as the reference patterns from a far greater number of clients to determine which objects should be prefetched. The actual prefetching, however, must be done by an agent closer to the client. Therefore, the server provides hints that assist this agent (either a proxy or client) in prefetching. Implementation of this model is complicated by the requirement for modifications at both the client or proxy side as well as the server side.

Given these options for caching and prefetching, we examine four different methods: passive proxy caching with unlimited storage; an active cache with local prefetching and unlimited storage; server-hint based prefetching alone; and an active cache with server-hint based prefetching and unlimited storage.

2.2 Bounding Analysis of Prefetching and Caching

We set upper bounds for each model by basing our simulations on some best-case assumptions. We assume that each method works with full knowledge of future events. Then we place certain restrictions on each method.

For passive caching, we assume that a previously accessed object that has not been changed is still available from the cache.

For local prefetching, since an object must be seen at least once before it can be predicted for prefetching, we assume that only the first access to an object will not be prefetched, and that all subsequent accesses will be successfully prefetched. We do not assume the use of additional information, outside the traced reference stream, that would allow the prediction of a future reference to a URL that has never been seen in the trace.

For server-hint based prefetching, we assume that prefetching can only begin after the client's first contact with that server. Because we assume a system with full knowledge of future events, without this restriction each server would schedule the transfer of each object to complete just before the object was requested. Provided there is enough bandwidth, this model would eliminate all communication latency. In such a system servers would suddenly transfer objects to clients with which they had never before been in contact. To provide a more realistic and useful bound on the performance of server-hint based prefetching, we assume that in order for a server to provide prefetch hints for a client, it must have been accessed by that client. In this *first-contact* model, upon the first contact from a client, a proxy will simultaneously prefetch all of that client's future requests from that server. So, for example, if you contact *www.cnn.com* on Tuesday, this model would assume that all of your requests to *www.cnn.com* for Wednesday, Thursday, and Friday would have been prefetched on Tuesday.

Even this first-contact model may be somewhat unrealistic. We investigated the effects of placing limits on the prefetching lead time, and on the amount of data prefetched simultaneously. To limit the amount of data prefetched simultaneously we place a threshold on the bandwidth that can be used for prefetching. After the first contact, subsequent requests are scheduled for immediate prefetch until this bandwidth threshold is reached or exceeded. The next request is then only scheduled to begin once some current prefetches have completed and the bandwidth being used has gone below the threshold. We varied, bandwidth and lead time independently and in combination.

Finally, to bound the performance of active caching using server-hint based prefetching and unlimited storage, we test if an object could be found in a passive cache. If this is not the case, we test if this object could have been successfully prefetched under the first-contact model.

3 Quantifying Bounds

We used trace-based simulation to quantify these bounds. To obtain traces, we modified the firewall Web proxy used by Digital Equipment Corporation to record all HTTP requests from clients within Digital to servers on the Internet. The traces ran from from 29 August to 22 September 1996. Each event in this trace stream represents one Web request-response interaction for a total of 24,659,182 events from 17,354 clients connecting to 140,506 servers. This proxy provided no caching or prefetching; it simply served as a method for crossing the corporate firewall.

To provide separate samples for comparison, we extracted three mid-week segments from the traces, each covering a Monday through Friday. We labeled these samples *Week 1* through *Week 3* for each work week and *all* for the entire trace stream. We also examined a subset of the trace data consisting only of clients in Palo Alto, CA, where the proxy was located. These samples are labeled *PA 1* through *PA 3* and *PA all*. The workload from this subset would be more representative of a smaller, more localized organization.

Since our main concern is the latency seen when retrieving a Web page, our simulations focus on requests that use the HTTP protocol and the *GET* method. We ignored events that failed for any reason (*e.g.* the connection to the server failed during data transfer). For all of our analyses, we assumed that query events and events with *cgi-bin* in their URL cannot be either prefetched or cached. This convention is used by most proxy caches.

In our simulations, we use the observed total and external latencies to estimate total (t), external (e) and internal ($i = t - e$) event latencies that would be seen if this request were to occur again. If our model says that a request could have been successfully prefetched or cached, then we use the internal latency as our estimate for the modeled event's new duration ($n = i$). Otherwise, we use the previously observed total latency ($n = t$). Given these values, we then quantify the fraction of the latency reduced as $(t - n)/t$. This approximation ignores the possibility that the proxy was transferring information to the client during its communication with the server. However, since we are looking for a lower bound on the latency of this event, which is the same as an upper bound on the latency reduction, we can ignore this possibility.

Using these approximations and the bounding models described in §2, our simulation steps through each event in the trace stream and determines whether it could have been cached or prefetched. We compare the distribution of event latencies seen under these bounding models with those in the original trace (measured without caching or prefetching), in order to compute the average latency reduction, for the workload represented in these traces.

3.1 Sources of Inaccuracy

We note several possible sources of inaccuracy for the results presented here. Our assumption that URLs

Table 1: Passive caching with unlimited storage.

Measurement	Week 1	Week 2	Week 3	All	PA 1	PA 2	PA 3	PA All
Total latency t	4.6	4.7	4.3	4.1	2.8	2.4	2.7	2.4
External latency e	3.6	3.6	3.2	3.2	2.6	2.2	2.4	2.1
New latency n	3.5	3.6	3.4	3.0	2.5	2.2	2.4	2.0
Percentage external latency e/t	79%	77%	75%	77%	90%	90%	88%	88%
Total latency reduction	24%	22%	22%	26%	12%	11%	11%	15%
Hit ratio	48%	47%	48%	52%	19%	20%	20%	28%
Cache size (GB)	23	26	27	88	1.3	1.2	1.1	4.5

Latencies are averages in seconds. *PA 1–3* and *PA all* represent work week 1 through work week 3 and the entire trace stream for the Palo Alto subset.

Table 2: Bounds on latency reductions from local prefetching.

Measurement	Week 1	Week 2	Week 3	All	PA 1	PA 2	PA 3	PA All
Percentage external latency	79%	77%	75%	77%	90%	90%	88%	88%
Total latency reduction	41%	38%	36%	45%	26%	22%	24%	33%

with queries and *cgi-bin* cannot be prefetched or cached might cause our upper bound on latency reduction to be less than the best possible latency reduction. An algorithm that is able to cache or prefetch such items could see greater latency reductions then those presented here.

Our traces lack last-modified timestamps for about half of the entries because many servers do not provide this information. This left us to use changes in response size as the only indicator of stale data for these requests. The result is a simulation that in some cases would simulate a cache hit when one should not occur, causing us to overestimate the potential for latency reduction.

Our models assume that the latency for retrieving a successfully prefetched or cached item from the proxy is the time of the original event minus the time for proxy-server communications ($t - e$). This assumes that there is little or no overlap between the server-to-proxy data transfer and the proxy-to-client data transfer. When this assumption is wrong, as it is for the larger responses and more distant clients, our simulation will overestimate the possible latency reduction.

4 Results

The goal of our simulations was to use the workload traced to quantify the four bounding models presented in §2.2. We first present the latency reductions for passive caching and active caching with local prefetching. We then examine the *first-contact* model with and without unlimited storage caching. We address the variation in latency reduction by examining the distribution for latency reduction for each event and for the different types of objects that were requested.

4.1 Passive Proxy Caching Performance

First, we simulated a passive caching proxy with unlimited storage. Table 1 shows the results. The first three rows show the averages for original total latency, external latency and the simulated latency for a passive caching proxy. The next row shows what fraction of the original total latency was contributed by the external latency. This ratio serves as a limit: if we could remove all external latency, then our average event latency would be reduced by this percentage. The last three rows show the percent of latency reduced by the simulated caching proxy, the cache hit ratio and the size of the cache that would be required to hold all cached data.

From this table we can see that passive caching is only able to reduce latency from 22%–26% (15% for a smaller organization), a far cry from the 77% (or even 88%) of external latency seen in the traces. Also, while the cache hit ratio ranges from 47%–52%(19%–28%), the latency reduction is only half of that. This implies that the majority of the requests that saw a cache hit are for objects smaller than the average event, which confirms a similar observation made by Williams *et al.* [15]. That study showed a weak inverse relationship between the likely number of accesses per unchanged response and the response size.

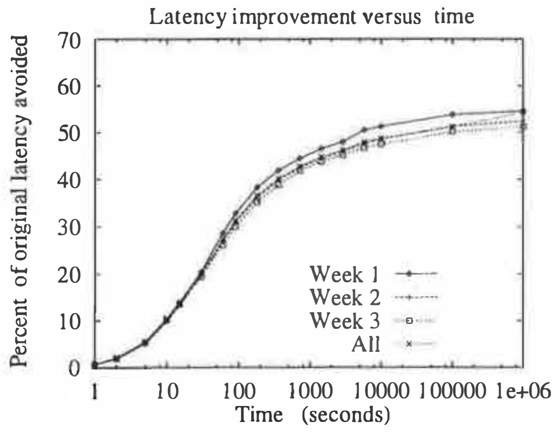
In Table 1, the latency reduction, hit ratio and cache size are larger for the entire-trace columns than for the single-week columns. This occurs because with the longer traces, there is a higher chance that given object will be referenced.

4.2 Local Prefetching

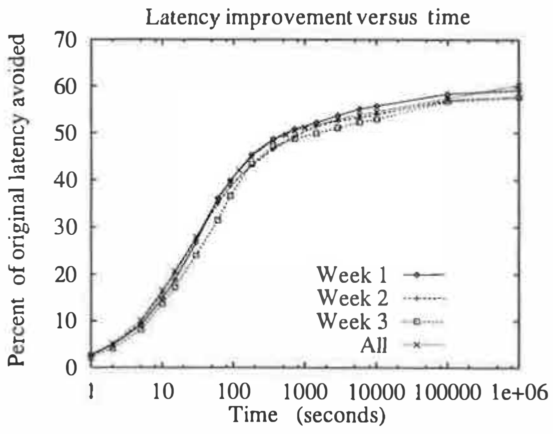
Next, we simulated prefetching based on locally-available information. Here, we assume that an object

Table 3: Results of server hint-based prefetching for an unlimited first-contact model.

Measurement	Week 1	Week 2	Week 3	All	PA 1	PA 2	PA 3	PA All
% external latency	79%	77%	75%	77%	90%	90%	88%	88%
Total latency reduction without caching	53%	51%	50%	53%	57%	56%	56%	58%
Total latency reduction with caching	58%	56%	54%	57%	59%	58%	57%	60%



(a) All clients in Digital



(b) Digital Palo Alto clients

Figure 2: Percentage of latency reduced versus how far in advance requests may be prefetched.

may be prefetched if and only if it has been seen before in the reference stream. This model provides a bound on active caching with local prefetching and unlimited storage. This model differs from the passive-caching model in that even if an object has changed (as indicated by a change in either the size or the last-modified timestamp), a subsequent access to that object can still be prefetched.

Table 2 shows that the latency reduction bound for local prefetching is almost double that for passive caching (see Table 1). The two results differ because the passive cache pays for a miss when an object changes; the high observed rate of change [3, 11] is what causes much of the poor performance of passive caching.

4.3 Bounds on Prefetching with Server Based Hints

To simulate server-hint based prefetching, we assume that prefetching can only begin after the client has contacted the server for the first time. To simulate a combination of caching and prefetching, our simulator first checks if an object is in the cache. If not, then the simulator uses the server-hint based model to determine if this object could have been successfully prefetched. Table 3 shows that this first-contact model, where all future requests are simultaneously prefetched upon first contact, will reduce the latency by a little more than half.

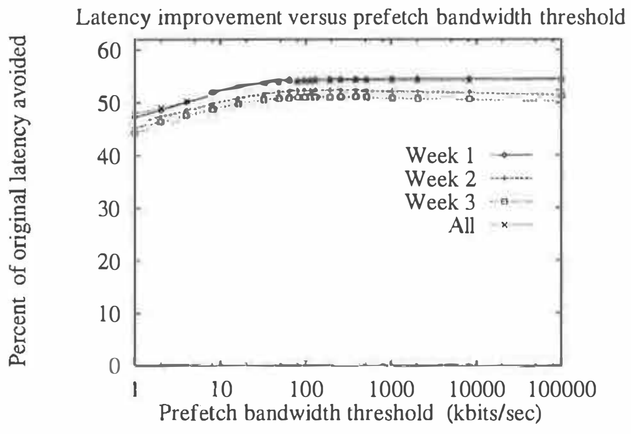
For a smaller, more centralized organization (as represented by the Palo Alto subsets), even though the external latency increases to 88% of the total latency, the improvement from server-hint based prefetching is only slightly better. In combination with an unlimited storage caching, server-hint based prefetching provides at best a 60% latency reduction.

4.3.1 Limiting the First-Contact Model

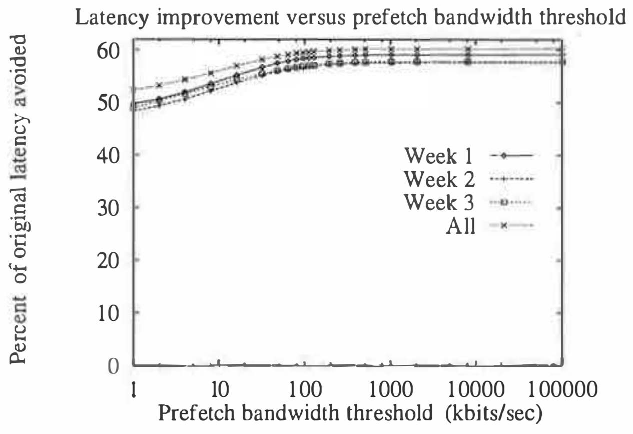
We modified our first-contact model to provide a more realistic bound by placing a limit on how far in advance of a request the data could be prefetched, also known as *prefetch lead time*. We then further modified our model to limit the amount of data prefetched simultaneously.

In order to examine the effects of limiting *prefetch lead time*, we modified our simulation to forget about contacts between client-server pairs that have been inactive for longer than a specified interval. This means that any subsequent request from the given client to the given server will be treated as a first contact. Figure 2 shows the results of varying prefetch lead times from one second to ∞ (represented in the figures as one million seconds). We note that for lead times below a few hundred seconds, the available reduction in latency is significantly less impressive.

To limit the amount of overlap between prefetch requests, we set a threshold on the bandwidth that can



(a) All clients in Digital



(b) Digital Palo Alto clients

Figure 3: Percentage of latency reduced versus bandwidth threshold.

used for prefetching. Figure 3 shows how this bandwidth limit affects the amount of latency reduction. At the left end of the graph, a small increase in the amount of bandwidth limiting prefetching significantly improves the reduction in latency. However, the total difference between unlimited simultaneous prefetching and sequential prefetching is no greater than 11%.

Figure 4 and Table 4 show the effects of varying both limits. The slope in Figure 4 along the axis for prefetch lead time shows that this parameter is the dominant factor. The relatively consistent slopes in the surface graphs imply the two parameters are relatively independent in their effects on latency. Therefore, the curves in Figures 2 and 3 are representative of the behavior of this model over variations in both parameters.

Table 5 shows the latency reduction seen by a server-hint based prefetching model with unlimited cache storage, for representative values of prefetch time and bandwidth available for prefetching. Comparing Table 5 with Table 4, we note that for a weak or moderately capable prefetching algorithm, the use of caching is especially important. For example, for a prefetching algorithm that can predict requests up to 3 minutes in advance, and has bandwidth threshold of 8 kbits/sec, caching will still offer an increase of approximately 11%. On the other hand, for an unbounded prefetching model, caching only offers an improvement of approximately 4%.

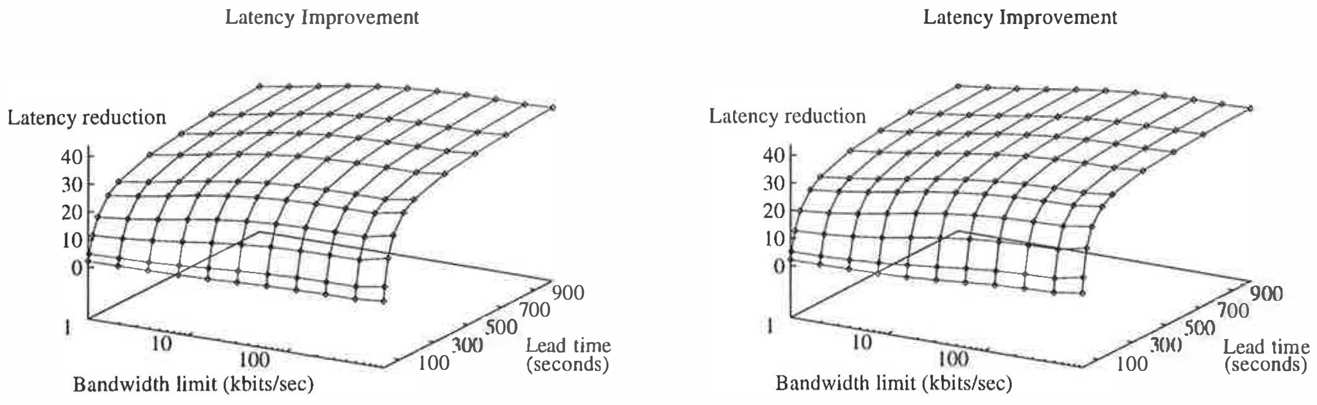
4.4 Reductions for Different Object Types

We examined how the type of object requested affects the latency reduced (object types were determined by an extension and if any, at the end of the URL path). Table 6 shows the results from caching and prefetching listed by object type. The category *NONE* represents URLs that had no extension, and the category *OTHER* represents objects with extensions other than those listed. The first two rows show the average for original total latency and the simulated latency. Rows 3 and 4 show the percentage of the total event stream that each type accounts for by event count and event duration, respectively. Rows 5 through 8 show the percentage of latency reduced by type for passive caching, local prefetching with unlimited cache storage, server-hint based prefetching without caching and server-hint based prefetching with unlimited storage caching.

This table shows that for the most part, the common types (*GIF*, *HTML*, *JPEG*, *CLASS*, *DATA*) all offer slightly above average latency reductions, while the less common or type-ambiguous requests offer significant less benefit. This result suggests that a cache which only kept objects of common types might make more effective use of its storage resources.

4.5 Variation in Latency Reduction

To examine the variation of latency reduction across separate events, Figure 5 shows histograms of the percent of latency reduced at each event for passive caching, local prefetching, server-hint based prefetching and server-hint based with unlimited storage caching. These graphs show a bi-modal distribution where an event either sees significant latency reductions (the peaks around 95% and 100%) or little to no reduction (the peak at 0%). What these distributions show is that under the models we have simulated one can expect a web request to either see minimal to no latency reduction or a significant



(a) All clients in Digital

(b) Digital Palo Alto clients

Figure 4: Reduction in latency for prefetching (bandwidth in kbits/sec, time in seconds). Note: the scale for the X axis is logarithmic.

Table 4: Selected latency reduction percentages for prefetching (bandwidth in kbits/sec, time in seconds).

b/w	Time	Week 1	Week 2	Week 3	All	PA 1	PA 2	PA 3	PA All
1	10	4.2%	3.9%	4.2%	4.3%	4.0%	5.1%	3.7%	5.6%
8	10	5.3%	4.9%	5.2%	5.3%	5.5%	6.6%	5.1%	7.0%
∞	10	9.8%	9.3%	9.6%	9.9%	13.8%	14.9%	12.9%	15.3%
1	60	16.1%	15.3%	14.9%	15.6%	18.7%	19.0%	16.7%	20.0%
8	60	20.8%	19.6%	19.1%	20.0%	24.7%	23.8%	21.4%	24.9%
∞	60	27.7%	26.1%	25.2%	26.5%	34.6%	33.2%	30.0%	33.9%
1	180	25.6%	24.0%	23.5%	24.4%	29.6%	27.7%	27.5%	29.5%
8	180	31.5%	29.4%	28.8%	29.8%	35.3%	33.0%	33.3%	34.7%
∞	180	37.4%	35.2%	34.2%	35.6%	43.6%	41.1%	41.7%	42.7%
1	∞	46.2%	44.3%	43.1%	46.7%	47.9%	46.3%	47.2%	50.1%
8	∞	50.4%	48.1%	47.0%	50.2%	51.7%	50.1%	51.2%	53.3%
∞	∞	53.4%	51.3%	49.9%	53.0%	57.1%	55.5%	55.8%	57.8%

reduction in the total latency.

5 Related Work

Many researchers have looked for ways to improve current caching techniques.

Padmanabhan and Mogul [12] described a server hint-based predictive model. In this model, the server observes the incoming reference stream to create a Markov model predicting the probability that a reference to some object A will be followed, within the next n requests, by a reference to some other object B (n is a parameter of the algorithm). On each reference, the server uses this model to generate a prediction of one or more subsequent references and sends this prediction as a hint to the client, including it in the response to the current reference. The client may then use this hint to prefetch an object if the object is not already in the client's cache. In their simulations, they estimated reductions of as much as 45%, but note that such techniques will also double the network traffic. Nevertheless, they show that a reasonable balance of latency reduction and network traffic increase can be found.

Bestavros *et al.* [1] have presented a model for the speculative dissemination of World Wide Web data. This work shows that reference patterns from a Web server can be used as an effective source of information to drive prefetching. They observed a latency reduction of as much as 50%, but only at the cost of a significant increase in the bandwidth used.

Williams *et al.* [15] presented a taxonomy of replacement policies for caching within the Web. Their experiment examined the hit rates for various workloads. The observed hit rates that range from 20% to as high as 98%, with the majority ranging around 50%. The

Table 5: Latency reduction percentages for both prefetching and caching (bandwidth in kbits/sec, time in seconds).

b/w	Time	Week 1	Week 2	Week 3	All	PA 1	PA 2	PA 3	PA All
1	10	26.2%	23.9%	23.5%	27.7%	15.0%	15.1%	13.6%	18.9%
8	10	26.8%	24.5%	24.0%	28.2%	16.2%	16.3%	14.7%	19.9%
∞	10	29.1%	26.8%	26.3%	30.2%	22.9%	23.0%	21.1%	26.0%
1	60	32.8%	30.6%	29.9%	33.6%	27.5%	27.4%	25.1%	30.6%
8	60	35.3%	33.1%	32.3%	35.8%	32.5%	31.4%	28.9%	34.4%
∞	60	38.9%	36.7%	35.7%	38.8%	40.8%	39.1%	36.0%	41.2%
1	180	39.3%	36.8%	35.9%	39.3%	36.8%	35.1%	35.0%	38.5%
8	180	42.4%	40.0%	38.8%	42.0%	41.4%	39.6%	39.7%	42.6%
∞	180	45.7%	43.4%	42.0%	44.9%	48.3%	46.2%	46.6%	48.6%
1	∞	53.5%	51.3%	50.0%	54.0%	50.2%	50.0%	49.9%	53.4%
8	∞	55.9%	53.6%	52.3%	55.9%	53.5%	53.3%	53.4%	56.1%
∞	∞	57.7%	55.5%	54.0%	57.4%	58.5%	58.1%	57.4%	59.9%

Table 6: Latency reduction for prefetching and caching by type of data requested (times in seconds).

Type	NONE	HTML	GIF	DATA	CLASS	JPEG	MPEG	OTHER
New latency	1.3	2.4	1.6	0.3	1.6	3.9	110.8	12.9
Original latency	2.5	4.3	3.3	1.5	3.8	7.2	151.7	17.0
Percentage by count	29.5%	12.1%	42.4%	0.2%	0.3%	11.1%	0.0%	5.4%
Percentage by time	17.8%	12.5%	34.3%	0.1%	0.3%	19.4%	0.4%	15.2%
Passive caching	26.7%	21.8%	34.5%	73.0%	40.4%	27.1%	4.5%	8.8%
Local prefetching	48.3%	43.7%	52.7%	79.9%	58.2%	45.8%	26.9%	24.3%
Server-hints without caching	49.3%	61.6%	62.5%	66.8%	64.4%	62.7%	27.5%	27.9%
Server-hints with caching	51.1%	61.5%	60.1%	79.9%	65.0%	58.4%	26.4%	27.6%

workload with the hit rate of 98% comes from a cache that is placed close to the server, rather than close to the clients, with the purpose of reducing internal bandwidth consumed by external HTTP requests. This workload addresses a different problem than that examined here. For their other workloads, our cache hit rates are reasonably comparable.

Several of the results found by other studies are close to or higher than the bounds resulting from our simulations. We note that these results are highly dependent on the workload, and on the environment modeled. One should take care in applying the results of any of these simulation studies, ours included, to a specific situation.

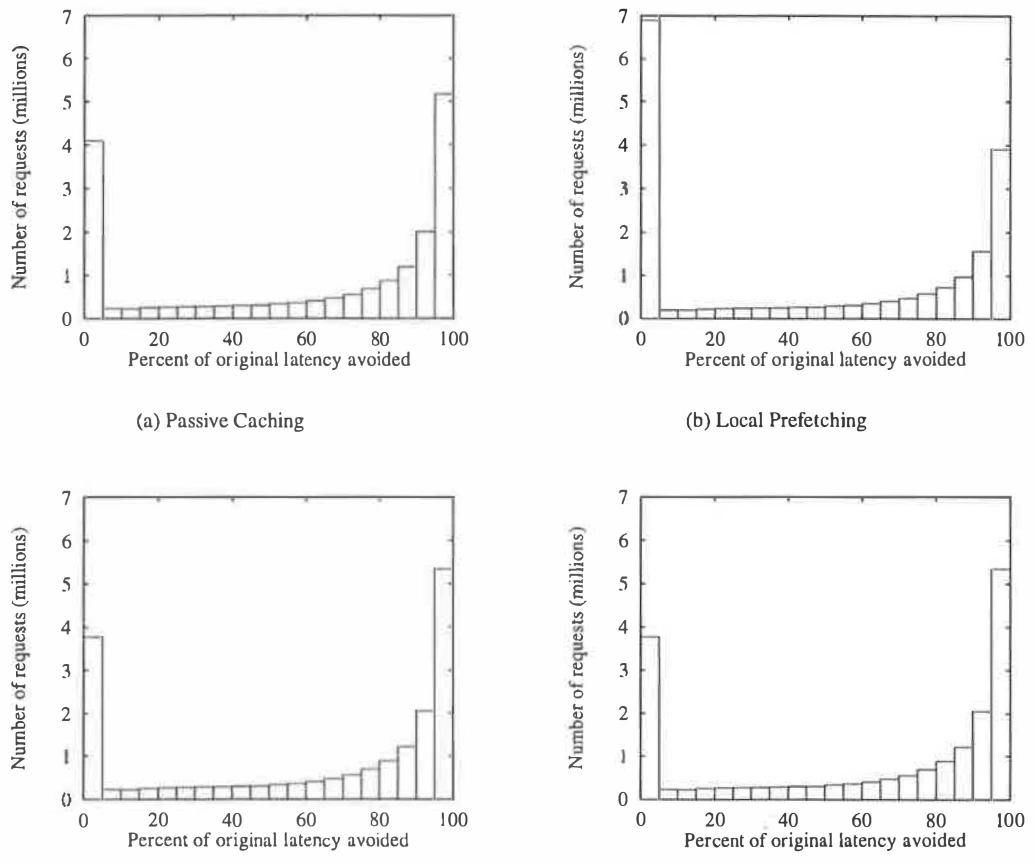
6 Summary and Conclusions

Using trace driven simulations we have explored the potential latency reductions from caching and prefetching. For the workload studied passive caching, with an unlimited cache storage, can reduce latency by approximately 26%. This disappointing result for passive caching is in part because data in the Web continually changes. On the other hand prefetching based on local information offers, saw a bound of approximately 41% reduction in latency. Adding server-hints increased this bound to approximately 57%.

We observed that prefetch lead time is an important factor in the performance of prefetching. Under more realistic constraints (prefetch bandwidth threshold of 8 kbits/sec, lead time of 3 minutes), the best we could hope for is a 35% reduction in latency. We also saw that the uncommon types of objects provided significantly less then average latency reduction.

Finally, we can make some observations based on comparing the results from these models with the total external latency. In the workload studied, 57% of the total latency is external latency that can be removed by caching or prefetching; 20% is external latency that cannot be removed, from events such as first contacts or queries and 23% is internal latency.

The results we present here serve to illustrate two conclusions. First, caching and prefetching can be effective in reducing latency in the Web. Second, the effectiveness of these techniques does have limits. The bounds produced by our simulations are for off-line algorithms that have full knowledge of the future. One would expect any on-line algorithm to be less effective. Again, we caution that these results are highly dependent on the workload and environment modeled. They should be applied with care. Nevertheless these results emphasize the need for improved prefetching techniques as well as additional techniques beyond caching and prefetching. These techniques might include wide-area replication [5], delta encoding [2, 11] and persistent TCP connections [10] which has been included in the HTTP/1.1 protocol [4].



(a) Passive Caching (b) Local Prefetching

(c) Server-Hint Based without Caching (d) Server-Hint Based with Caching

Figure 5: Histograms of the percentage of latency reduced at each event.

Acknowledgments

We are grateful to Digital Equipment Corporation for their support of this work and the use of their facilities, the Office of Naval Research for their support, Prof. Mary G. Baker for her kind guidance, Dr. Richard A. Golding for his insightful comments and enthusiasm, Prof. Pei Cao for her comments and input on this work, Carlos Maltzahn and Dr. Kathy Richardson for their work with the Squid proxy, Dr. Lawrence Brakmo for his input and comments on early portions of this work, Randal Burns, Tracey Sconyers and the other members of the concurrent systems group that provided comments on this work, and Suzan Fry.

Availability

Additional details on these traces, and the traces, are available from:

ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html

References

[1] A. Bestavros and C. Cunha, "A prefetching protocol using client speculation for the WWW," Tech. Rep. TR-95-011, Boston University, Department of Computer Science, Boston, MA 02215, Apr. 1995.

[2] R. C. Burns and D. D. E. Long, "Efficient distributed backup with delta c1ompression," in *Proceedings of the 1997 I/O in Parallel and Distributed Systems (IOPADS'97), San Jose, CA, USA*, Nov. 1997.

[3] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul, "Rate of change and other metrics: A live study of the world wide web," in *Proceedings of*

First USENIX Symposium on Internet Technologies and Systems, Dec. 1997.

[4] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2068, HTTP Working Group, Jan. 1997.

[5] J. Gwertzman and M. Seltzer, "The case for geographical push caching," in *Fifth Annual Workshop on Hot Operating Systems*, (Orcas Island, WA), pp. 51–55, IEEE Computer Society, May 1995.

[6] J. Gwertzman and M. Seltzer, "World wide web cache consistency," in *Proceedings of the USENIX 1996 Annual Technical Conference*, (San Diego, CA), pp. 141–152, USENIX, Jan. 1996.

[7] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felton, G. A. Gibson, A. Karlin, and K. Li, "A trace-driven comparison of algorithm for parallel prefetching and caching," in *Proceedings of Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 19–34, USENIX, October 1996.

[8] T. M. Kroeger and D. D. E. Long, "Predicting filesystem actions from prior events," in *Proceedings of the USENIX 1996 Annual Technical Conference*, USENIX, January 1996.

[9] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *Proceedings of USENIX 1997 Annual Technical Conference*, USENIX, January 1997.

[10] J. C. Mogul, "The case for persistent-connection HTTP," in *Proceedings of the 1995 SIGCOMM*, pp. 299–313, ACM, Sept. 1995.

[11] J. C. Mogul, F. Douglis, A. Feldmann, , and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," in *Proceedings of the 1997 SIGCOMM*, (Cannes, France), pp. 181–194, ACM, Sept. 1997.

[12] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *Computer Communications Review*, vol. 26, pp. 22–36, July 1996.

[13] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[14] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, pp. 771–793, September 1996.

[15] S. Williams, M. Abrams, C. R. Standridge, C. Abdulla, and E. A. Fox, "Removal policies in network caches for world-wide web documents," in *Proceedings of the 1996 SIGCOMM*, pp. 293–305, ACM, July 1996.

The Measured Access Characteristics of World-Wide-Web Client Proxy Caches

Bradley M. Duska, David Marwood, and Michael J. Feeley
Department of Computer Science
University of British Columbia
{bduska,marwood,feeley}@cs.ubc.ca

Abstract

The growing popularity of the World Wide Web is placing tremendous demands on the Internet. A key strategy for scaling the Internet to meet these increasing demands is to cache data near clients and thus improve access latency and reduce network and server load. Unfortunately, research in this area has been hampered by a poor understanding of the locality and sharing characteristics of Web-client accesses. The recent popularity of Web proxy servers provides a unique opportunity to improve this understanding, because a small number of proxy servers see accesses from thousands of clients.

This paper presents an analysis of access traces collected from seven proxy servers deployed in various locations throughout the Internet. The traces record a total of 47.4 million requests made by 23,700 clients over a twenty-one day period. We use a combination of static analysis and trace-driven cache simulation to characterize the locality and sharing properties of these accesses.

Our analysis shows that a 2- to 10-GB second-level cache yields hit rates between 24% and 45% with 85% of these hits due to sharing among different clients. Caches with more clients exhibit more sharing and thus higher hit rates. Between 2% and 7% of accesses are consistency misses to unmodified objects, using the Squid and CERN proxy cache coherence protocols. Sharing is bimodal. Requests for shared objects are divided evenly between objects that are narrowly shared and those that are shared by many clients; widely shared objects also tend to be shared by clients from unrelated traces.

1 Introduction

The growing popularity of the World Wide Web is placing tremendous demands on the Internet and is increasing the importance that the Internet function effectively.

The scale problems facing the Web can be addressed on three fronts. The first is to scale Web servers to handle the increasing demands being placed on them. The second is to ensure that the Internet itself will scale by continuing to increase its capacity and by deploying new network technologies. The third is to focus on the clients: Web browsers and a hierarchy of *proxy servers* to which they may be connected.

Client-side solutions such as caching and prefetching are attractive because they improve the performance of both client and server. Caching, in particular, aids scaling by reducing the amount of data requested from servers and transferred through the network.

The potential benefit of client-side caching has drawn considerable research and practical attention. A general framework is emerging that organizes client-side caches as a hierarchy. At the bottom are Web browser caches. Browsers can be configured to direct requests to a nearby proxy server, which then provides the second level of caching; some proxy caches are themselves connected to a third-level cache. The highest-level cache is typically a cooperative cache that connects lower-level caches to each other so that a miss in one cache can be satisfied by one of its siblings. This cooperation can be achieved either by multicasting misses to siblings [7, 5, 17] or by maintaining a shared directory of cache contents [9, 18]. Caching can also be provided by geographically distributed caches to which servers push popular data [12].

The promise of client-side caching has been demonstrated by several research prototypes and by some large-scale experiments in higher-level Web caching [5]. To date, however, researchers have had little data about client access patterns on which to base their designs and with which to assess them. There are two main reasons for this lack of workload information. First, the nature of Web clients continues to change rapidly and thus workload characterization is a moving target. Second, until recently, there has been little machinery in place for collecting data from a significant sample of Web clients.

Workload characterization is important because caching and prefetching are techniques that exploit specific access properties to improve performance: temporal locality and sharing in the case of caching, and spatial locality in the case of prefetching. Filesystem research, for example, has relied heavily on studies of Unix filesystem workloads that show that files tend to be small, accessed sequentially from beginning to end, read more than written, and rarely shared concurrently [3, 1, 19].

1.1 Key Issues for Cache Design

The lack of Web-client workload information has left a number of crucial cache-design issues poorly understood. The analysis presented in this paper addresses the following four sets of design issues.

Cache Size and Hit Rate First, what hit rate can we expect for a given cache configuration; that is, how does hit rate depend on the number of clients connected to the cache and how does the number of clients effect cache size?

Configuration of Cache Hierarchy Does a cache hierarchy make sense and if so, how many clients should be connected to each level, how many levels should there be, and how should higher-levels of the hierarchy be connected to each other?

Cache Coherence Is the coherence protocol currently used by most proxy caches (i.e., Squid [5] and CERN [14]) causing unnecessary cache misses?

Sharing How much sharing is there; are all Web objects shared equally; does sharing increase with the number of clients; and how does sharing within a group of clients (e.g., a company or a university) compare to sharing among unrelated clients?

1.2 Summary of Our Results

This paper presents our analysis of the client request stream collected from a sample of second- and third-level proxy caches located throughout the Internet. We collected twenty-one days of access traces from each proxy, totaling 23,700 clients and 47.4 million accesses. The following list summarizes our findings, discussed in detail in Section 3.

- Second-level cache hit rates vary from 24% to 45%; a higher request rate yields a higher hit rate. The NLANR third-level cache has a lower hit rate of 19% due to the expected filtering of locality and sharing from the request stream by lower-level caches.
- Ideal cache sizes ranged from 2 to 10 GBs, depending on client population size. Roughly 1-GB of cache is needed for each 70,000 to 100,000 requests/day (35,000 requests/day for small populations). A trace whose client population is artificially reduced, however, requires a somewhat larger cache.
- Using the Squid and CERN cache coherence protocol, 2% to 7% of requests are consistency misses to up-to-date cached objects, requests that would otherwise be hits.
- 85% of cache hits are due to sharing between clients. Sharing hit rates range from 20% to 38%; a higher request rate yields more sharing.
- Requests for shared objects account for up to 71% of all requests; but, only 15% to 24% of requested objects are shared (only half of these requests are hits due to first-time requests and consistency misses).
- Sharing is bimodal. Requests for shared objects are divided evenly between objects that are narrowly shared and those that are shared by many clients; widely shared objects also tend to be shared by clients from unrelated traces.

2 Methodology

This section details the methodology of our study. We begin with a discussion of client-side Web caching and issues for collecting access traces. We then provide a detailed description of the trace data we collected. Finally, we describe the trace-driven cache simulator we built to evaluate this data and we discuss its validation.

2.1 Collecting Access Traces from Proxy Servers

The main reason that little is known about Web client access patterns is that collecting information about these accesses is difficult. While it is easy to collect access data from a Web server, the accesses to a particular server shed little light on overall client access patterns. Web browsers, on the other hand, are not a practical source of data because of the logistical complexities of collecting data from a sufficiently large number of users. A feasible alternative is to collect data from Web proxy servers to which browsers

Trace	Collection Period	Number of Clients	Client ID Preserved	Requests (Millions)	Maximum Simulated Cache Size (GBs)
HAN	Jun 17 – Jul 7, 1997	1858	full period	5.28	14
KOR	Apr 20 – May 10, 1997	2247	full period	3.19	8
DEC	Aug 29 – Sep 18, 1996	16,663	full period	21.47	*unlimited*
GMCC	Jun 4 – Jun 26, 1997	953	full period	1.36	4
AU	Jun 4 – Jun 24, 1997	310	one day	1.86	6
UU	May 1 – May 21, 1997	990	full period	1.59	4
NLANR	Jun 3 – Jun 27, 1997	711	one day	12.65	8

Table 1: Summary of proxy-server access traces.

can optionally be connected. It is only very recently, however, that Web proxies have been deployed widely enough to provide a sufficiently large and diverse sample of client accesses.

Web proxies are found in most corporate organizations that use a *firewall* to protect their internal network from the vagaries of the Internet. To access the Web from within such a protected domain, Web browsers are configured to direct all outgoing Web-data requests to a designated proxy machine. The proxy forwards requests between the protected corporate network and the outside Internet.

A proxy can also act as a *second-level* Web cache (the Web browser being the first-level). In this configuration, proxies are becoming popular in unprotected environments such as universities due to caching benefits alone.

It is typically a simple matter to configure a proxy to record all object requests it receives from its client browsers. Access traces can also be collected from *third-level* proxies, whose clients are collections of second-level proxy servers. Third-level proxies were first suggested by the Harvest project [7].

Finally, we must ensure that any use of Web client access traces does not compromise user privacy. A client request includes the requesting host's IP address, which may identify the user, and the requested object's URL. Privacy concerns prohibit exposing information that identifies users with the objects they access. Our solution was to pre-process all traces used in our study to disguise the requesting host's IP address using a one-way function that permits us to compare two disguised addresses for equality while protecting user privacy.

2.2 Web Data Access Traces

We have collected access traces from the following seven proxy servers distributed throughout the Internet. Some of these traces are publicly available and others have been provided to us directly.

- The University of Hannover, Germany (HAN)
- The Nation Wide Caching Project of Korea (KOR)
- Digital Equipment Corporation (DEC)
- Grant MacEwan Community College, Alberta, Canada (GMCC)
- A major American University that has chosen to remain anonymous (AU)
- Utrecht University, the Netherlands (UU)
- The National Laboratory for Applied Network Research (NLANR)

Table 1 summarizes the traces, listing: the data collection period, the number of client hosts making requests, whether a client's disguise was preserved across days, the total number of requests, and the maximum simulated cache size. Every trace contains twenty-one days of Web access; some traces, however, have slightly longer collection periods because data was not available for all days in the period.

All of these organizations are using the *Squid* proxy server (and cache) [5], a derivative of the Harvest research system [7]. The DEC trace was collected by Digital using a specially instrumented cacheless version of Squid; the trace is available via anonymous ftp [21]. NLANR makes several traces available [8]; we selected the *Silicon Valley* trace, which includes caches from the San Francisco Bay area and from overseas. The KOR traces are from the gateway that connects Korea to the Internet. We used a modified version of this trace that excludes a large third-level cache, because second-level cache characteristics are more interesting to our study. The remaining traces are from proxies serving the general computing community of various colleges and universities.

The traces capture all Web user requests except those satisfied by browser caches or directed to the interal network thus bypassing the proxy. Each trace entry includes the request time, the disguised client IP address, and the requested object's URL and size. All traces but DEC also include the *cache action* taken by the cache (e.g., hit, miss, etc.). DEC, instead, includes the object's *last-modified time* as reported by the Web server that holds the object.

Unfortunately, the technique used to disguise client identity in the AU and NLANR traces does not preserve a unique disguise for a client for the entire collection period. Instead, in these two traces clients are assigned a new disguise every day. As a result, a client that makes requests on two different days appears to our simulator as if it were two different clients. We have taken a conservative approach to dealing with this limitation. The client count for AU and NLANR in Table 1 lists the maximum number of clients making requests on any given day, an underestimate of the actual number for the entire trace. In addition, we exclude AU and NLANR from our analysis of sharing presented in Section 3.4.

2.3 Simulator

To conduct our analysis, we built a trace-driven proxy cache simulator, called SPA. SPA faithfully simulates the collected traces at a cache size and request rate different from the original proxy.

To simulate a different request rate, a trace is first *reduced* by extracting all of the requests made by a randomly-chosen subset of clients, such that the remaining clients produce the desired request rate.

To simulate a different cache size from the original proxy, SPA follows a simplified version of the replacement and coherence policies used by the Squid proxy-cache version 1.1 [5] and appropriately configured versions of CERN [14]. The replacement policy is a variant of LRU.

The cache coherence protocol assigns a *time to live* based on either a configurable portion of the last-modified time or a default if no value is supplied. Expired objects remain cached. When an expired object is requested, the proxy sends an *if-modified-since* request to the server and receives a new copy from the server only if the object was actually modified. The results of certain requests such as dynamic scripts (e.g., cgi) and Web query forms are never cached.

For all but the DEC trace, SPA infers the time to live from the cache consistency operations of the original proxy. This inference is accurate only up to the cache size of the original proxy, because a larger SPA cache would hold objects not cached by the orginal proxy and thus the trace would contain no consistency information for these objects. The **Maximum Simulated Cache Size** in Table 1 shows the maximum size we simulated, a size not larger than the original proxy cache size.

Unlike the other traces, DEC includes the expiry and last-modified times returned by the server, which can be used directly to calculate time to live (i.e., no inference is necessary). Where these times are not supplied in the DEC trace, SPA sets the last-modified time and time to live according to the default Squid cache coherence protocol.

We validated SPA by simulating each trace with a cache size equal to the proxy's actual size. We then compared the simulated and actual hit rates and byte hit rates. DEC was excluded, because its proxy did not include a cache. In every case, the simulated hit rates were slightly lower than the actual hit rates; the average error was 3% (i.e., a hit rate difference of around 1.5%) and all errors were less than 4.5%. We believe the reason for the slight discrepancy is that the actual proxy caches were slightly larger than what we simulated.

We validated our trace-reduction scheme in two ways. First, we compared each reduction's per-client request rate to confirm that reducing a trace did not significantly change per-client request rate; average variance was 5% and maximum was 16%. Second, we measured the hit-rate variance among different versions of the same reduction, by computing 30 versions of the 5% GMCC reduction (i.e., each had a request rate that was 5% of the original). We simulated the 30 reduced traces and measured standard deviations of 1.9% hit rate and of 3.6% byte hit rate, even though each of the 30 reductions included a different randomly-chosen subset of the original client population.

3 Analysis

This section summarizes our analysis in five parts. First, we examine the relationship between cache size and hit rate. Second, we explain how increasing request rate increases hit rate. Third, we examine the impact of cache coherence on hit rate. Fourth, we provide a detailed analysis of Web-client sharing. Finally, we discuss how these workload characterizations impact cache design.

3.1 Cache Size and Hit Rate

The first question we address is: what is the hit rate and how does cache size impact hit rate?

Figure 1 shows each trace's simulated hit rate as cache

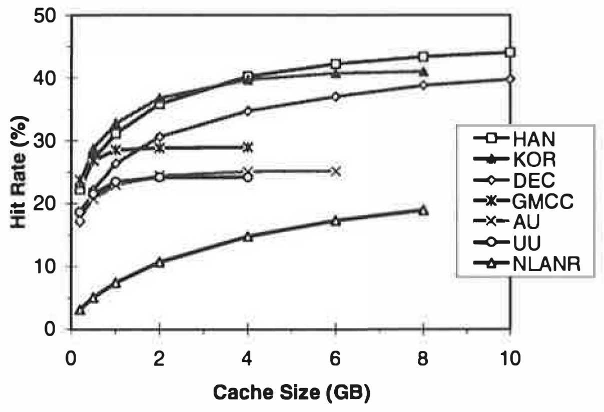


Figure 1: Cache hit rate for each trace as a function of cache size.

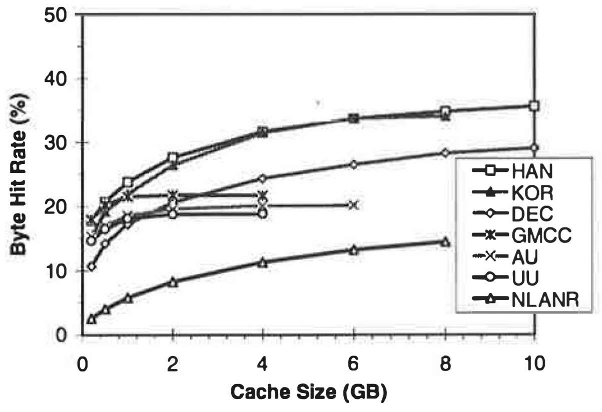


Figure 2: Cache byte hit rate for each trace as a function of cache size.

size is varied; hit rate is the ratio of cache hits to total requests. A trace's line on the graph stops at its maximum simulated cache size.

Figure 2 shows the *byte* hit rate for the same simulations; byte hit rate is the ratio of the total number of bytes supplied by the cache to the total number of bytes requested. The two graphs have the same shape but hit rates are roughly one third larger than byte hit rates, because smaller objects tend to have a slightly higher hit rate than larger objects.

Figure 1 shows that the hit rate of the NLANR third-level cache is considerably lower than that of the second-level caches. This lower hit rate is expected and is a common feature of higher-level disk caches [23]. The lower-levels of a cache hierarchy act as a filter, removing temporal locality and lower-level sharing from the reference stream. The resulting references received by the higher-level cache consist only of the lower-level caches' *capacity misses*, *consistency misses*, and first-time requests (i.e., references to objects that have been evicted, have expired, or have never been requested). Nevertheless, the NLANR third-level cache achieves a non-trivial 19% hit rate.

The two figures also show that most of the second-level cache hit rates level off at cache sizes smaller than 10 GBs. Hit rates for DEC and HAN continue beyond the largest cache size shown in Figure 1. DEC hit rates reach 41.1% for a 20 GB cache and increase very slowly to 42.1% for a 100 GB cache. HAN hit rates increase slightly to 44.7% for a 14 GB cache. These graphs seem to indicate that the largest cache needed to eliminate most capacity misses is dictated by a cache's request rate. For smaller traces, a cache size of 1-GB per 35,000 requests/day is adequate and for the larger traces, 1-GB per 70,000 to 100,000 requests/day is needed. We will see in the next section, however, that the relationship between request rate and cache size is not quite this straightforward.

3.2 Request Rate and Hit Rate

Figure 1 shows that, for the second-level traces, hit rate increases with request rate. The reason for this correlation is that a higher request rate causes more sharing and increases the number of hits an object receives before it expires.

The correlation between request rate and hit rate, however, is not perfect. There are two exceptions. First, DEC has the highest request rate but its hit rate is slightly lower than HAN and KOR. Second, GMCC's request rate is lower than UU and AU, but its hit rate is higher. Furthermore, the relationship between request rate and hit rate is not linear, as is seen by KOR, AU, and HAN. KOR's request rate is 1.7 times higher than AU and 1.7 times lower than HAN, but KOR's hit rate is twice AU's and only slightly smaller than HAN's. We examine this relationship in greater detail throughout the remainder of this section.

Reduced Traces

To gain a better understanding of the impact of request rate on hit rate, we examined the hit rate of each trace at various artificially reduced request rates.

To conduct this experiment, we produced a set of seven reduced traces for each of the original traces at 5%, 10%, 20%, 40%, 60%, 80%, and 100% of the original request rate, as described in Section 2.3. We then simulated the behavior of each of the 49 reduced traces.

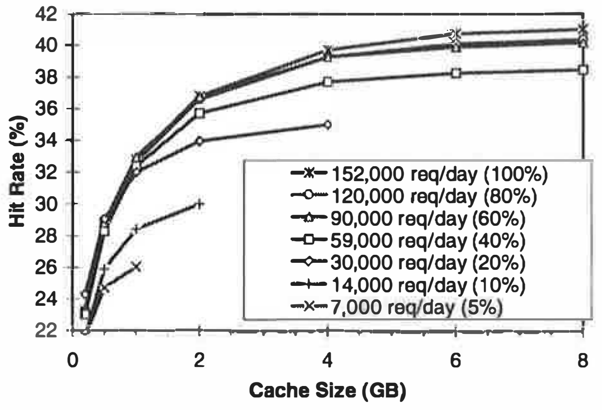


Figure 3: Cache hit rate for KOR as a function of cache size for a range of request rates.

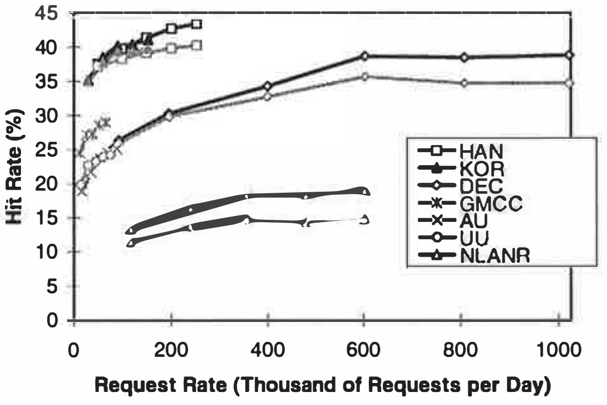


Figure 4: Cache hit rate for each trace (and selected cache sizes) as a function of request rate; generated by reducing the number of clients. Light colored lines are 4 GB caches. Dark colored lines are 8 GB caches.

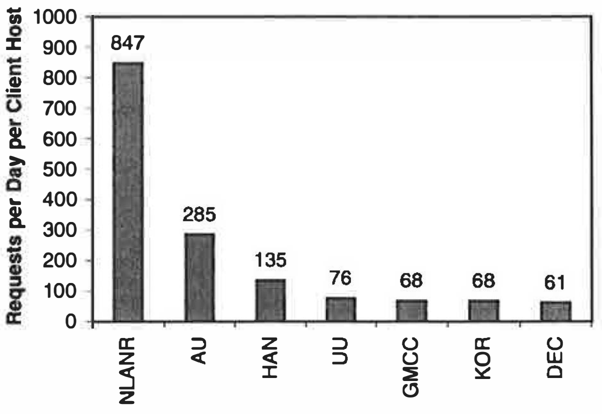


Figure 5: Average number of requests a client host makes per day.

Figure 3 shows the hit rate for the seven KOR reductions as cache size is varied. The request rate of each reduction is shown in the legend. The top line of the graph, 152,000 requests/day, represents the full trace and is the same as the KOR line in Figure 1. The lines for lower request rates stop when the cache is big enough to hold every object referenced; increasing cache size beyond this point would have no impact on hit rate so we do not show the line.

For each KOR reduction in Figure 3, hit rate increases with request rate, just as we saw in Section 3.1 when comparing the request rates of different traces. The details of the two relationships, however, are somewhat different. Figure 3 shows that reducing KOR's request rate does not significantly reduce desired cache size. For example, the hit rates for the 60%, 80%, and 100% reductions all appear to level off at the same cache size, around 6 GBs. We believe that the reason cache size does not decrease as rapidly as hit rate is that clients in the same cache share common interests. As a result, a 40%-reduced client population requests virtually the same set of shared objects as the full population. This characteristics of Web sharing is discussed in detail in Section 3.4.

Figure 4 shows the relationship between hit rate and request rate for all traces. For each trace we show hit rates for two size caches: 4-GB caches are shown using light-colored lines and 8-GB caches are shown with dark lines. Some traces do not have an 8-GB line, because their maximum simulated cache size is less than 8-GBs. This graph confirms that, like KOR, hit rate increases with request rate for all traces. Notice that in the 4-GB DEC run, hit rate decreases slightly with request rate due to thrashing.

Requests per Client

We now extend our analysis to determine how a cache's hit rate is affected by the number of clients hosts that are connected to it.

Figure 5 shows the average number of requests/day per client for each trace. These averages were computed by dividing each trace's request count by its total number of clients and then dividing by twenty one. The client counts for NLANR and AU shown in Table 1 are underestimated, because client disguises change from day to day, as discussed in Section 2.2. As a result, the values presented in Figure 5 for these two traces are upper bounds.

From Figure 5 we see that, for most of the second-level caches, clients made an average of 70 requests/day. AU and HAN have higher request rates, possibly because they

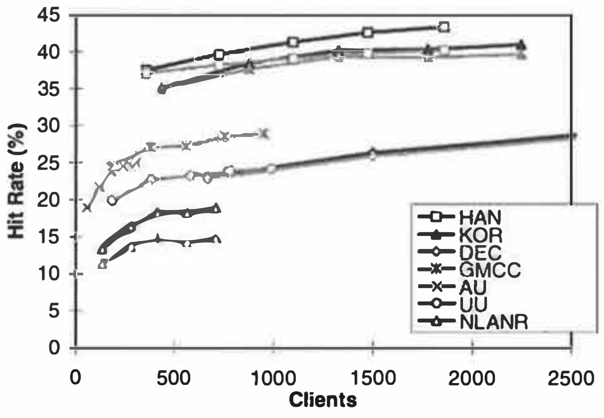


Figure 6: Cache hit rate for each trace as a function of the number of client hosts. Light colored lines are 4 GB caches. Dark colored lines are 8 GB caches.

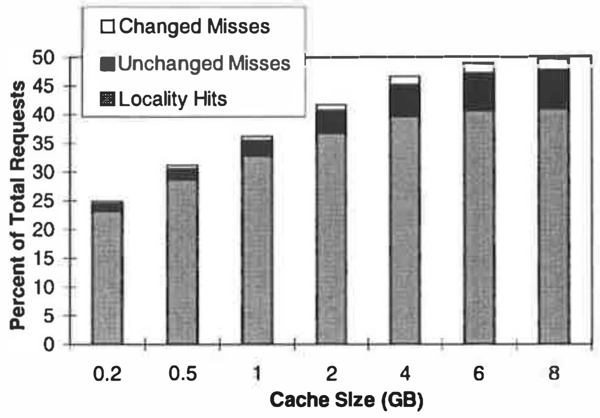


Figure 8: Portion of requests resulting in consistency misses to changed and unchanged objects in the KOR trace.

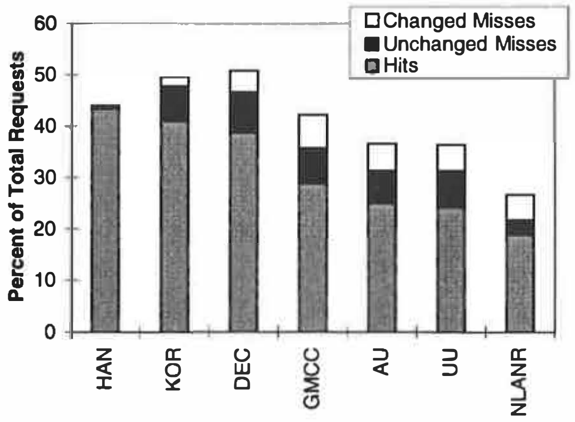


Figure 7: Portion of requests resulting in consistency misses to changed and unchanged objects in each trace.

have more multi-user hosts than the other traces. As expected, clients of the NLANR third-level cache have a much higher request rate than the second level caches (i.e., 847 request/day), because these clients are other caches and not browsers.

Figure 6 restates Figure 4, changing the x-axis from request rate to client count; the DEC line on this graph would extend to 16,700 clients. Notice that the shape of some of the lines has changed from Figure 4 to 6 due to the variety in the per-client request rates from different traces. The differences between these two graphs suggest that while client-count information is interesting, request rate is a better metric of cache performance.

3.3 Web Cache Coherence Protocols

We now examine how hit rate is impacted by the cache coherence protocol used by the Squid and CERN proxy caches; this protocol was described in Section 2.3.

Figure 7 shows each trace's maximum hitrate from Figure 1 with two additional bars, one labeled **Unchanged Misses** and the other **Changed Misses**. The total of these two bars is the consistency miss rate (i.e., the percentage of requests that found expired objects in the cache). An unchanged miss is a request for an expired but unmodified cached object. In this case, the coherence protocol requires an *if-modified-since* request to the object's server to verify that the object is still valid and to update its expiry time. In contrast, a changed miss is a request for a cached object that had changed at the Web server.

The height of the **Unchanged Misses** bar is a measure of the inefficiency of the coherence protocol. This protocol uses the time since an object was last modified to predict its next modification time and thus set an expiry time for the object. If an object expires before it actually changes, an unchanged miss results. Figure 7 shows that unchanged misses account for between 2% and 7% of all references. This percentage represents the hit rate improvement possible for coherence protocols that do a better job of predicting object expiry (e.g., [13] proposes one such protocol).

Figure 8 shows the same information as Figure 7 for the KOR trace and a variety of cache sizes. We now see that as cache size increases, the consistency miss rate grows much more quickly than hit rate. This faster growth is explained by the fact that a larger cache holds more expired objects and thus some capacity misses of a smaller cache become

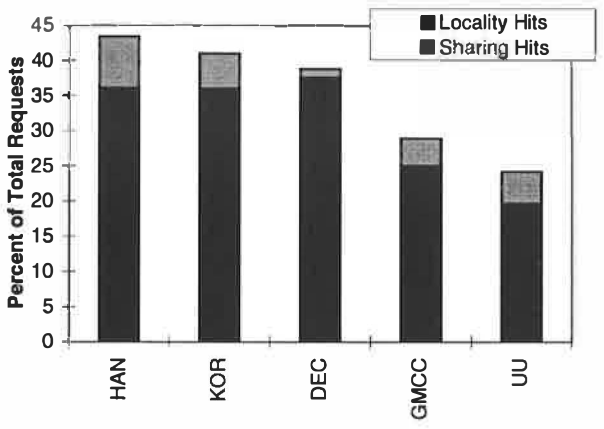


Figure 9: Hit rate divided into hits due to sharing and due to locality of a single client.

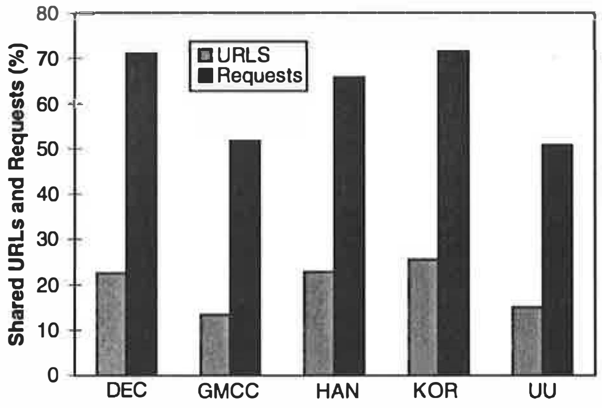


Figure 10: The percent of a total URLs in a trace requested by two or more clients and the percent of total requests to these shared objects.

consistency misses in the larger cache.

3.4 Sharing

We begin our discussion of sharing by dissecting the hit rates we have already presented to show how many hits are due to sharing. We then examine how requests are distributed to shared objects and what portion of these requests are actually hits. Finally, we examine the sharing among clients from unrelated traces.

All simulation results presented in this section use the maximum cache size for each trace as shown in Table 1, or 8 GB for DEC.

A fundamental limitation of the available trace data is that requests are identified with client hosts and not with users. Our analysis thus includes some *false sharing* (i.e., due to users who move from host to host) and *missed sharing* (i.e., due to hosts that serve multiple users).

The Sharing Hit Rate

To determine how may hits result from sharing, we modified our simulator to count locality and sharing hits separately. Any hit that could have been satisfied by a sufficiently large browser cache is classified as a locality hit; all other hits are shared hits. The modified simulator detects shared hits using a post-processing phase that conceptually examines every hit, checking backward in the input trace for previous references to the same object. A hit is shared if and only if a previous reference was made by a different client and all intervening references to the object are also hits by other clients.

Figure 9 shows the hit rates from Figure 1 divided into **Locality Hits** and **Sharing Hits**. The figure includes data for only five of the seven traces. NLANR and AU are excluded because the daily changing of client disguise in these traces makes it impossible to distinguish sharing from locality, as discussed in Section 2.2.

The most important feature of Figure 9 is that sharing is high and increases with client population and request rate. In every trace, sharing accounts for at least 85% of all hits. Furthermore, traces with higher request rates also have more sharing. For example, DEC, the trace with the highest request rate, also has the highest sharing hit rate at 38%. Notice that sharing rate is more closely correlated with request rate than hit rate was; DEC's hit rate, for example, was not the highest of the traces.

In contrast, locality hits do not increase with request rate. All traces have roughly the same locality hit rate of 5% (the exception is DEC at 1.5%). In other words, clients from both small and large populations request roughly the same proportion of non-shared objects, even though there is more sharing in a large population. It thus appears that adding a new client to a cache turns some of the misses of other clients into hits but does not change locality hits into shared hits.

Distribution of Requests to Shared Objects

We now examine how shared requests are distributed to Web objects. Figure 10 shows two bars for each of the five traces that preserve client identity. The first bar indicates the percentage of objects that are requested by multiple clients and the second bar indicates the percentage of requests that ask for one these shared objects. Notice that the shared request rate is much higher than the shared hit rate shown in Figure 9, because not all requests to shared

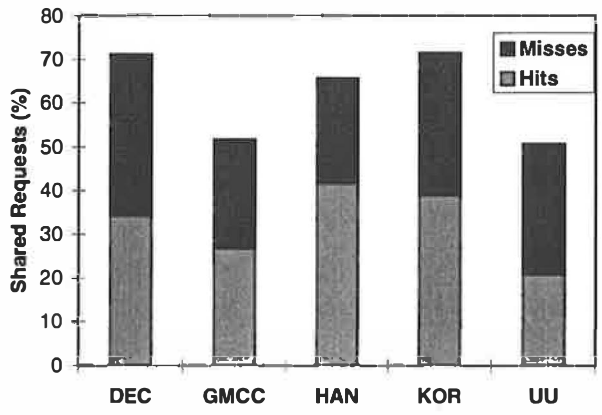


Figure 11: Accesses to shared objects divided into those that hit in cache (shared hits) and those that miss (i.e., first-time access or consistency or capacity misses).

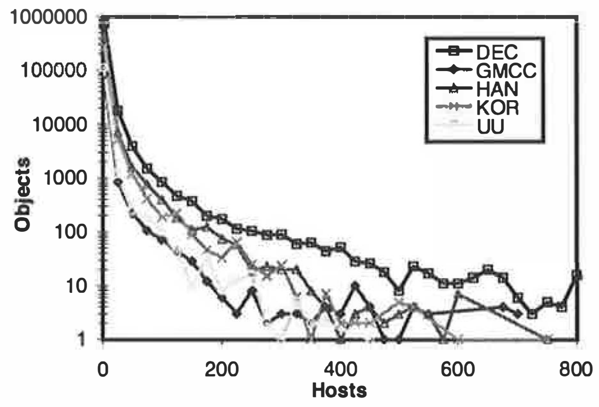


Figure 12: Histogram showing the distribution of Web object popularity (represented with lines in order to show multiple traces on the same graph). The y-axis is a log-scale of the number of object and the x-axis in the number of hosts that request the object.

objects are hits.

Figure 11 provides additional detail for the **Shared Requests** bar in Figure 10. The total height of each trace's request bar is the same in both figures. Figure 11, however, indicates the number of shared requests that hit in the simulated cache. The remainder of these requests are misses due to first-time accesses, consistency misses, and some capacity misses. In most cases, roughly half of the requests are hits, though HAN has slightly more hits than misses and UU has slightly less.

The key feature of Figure 10 is that while a very large portion of accesses are to shared objects (71% for DEC), only a small portion of objects are shared (23% for DEC). Notice further that the ratio between object count and request count is roughly the same for all traces, thought the actual sharing is lower for the smaller client populations.

Figures 12 and 13 provide additional detail about how requests are distributed among shared objects. Figure 12 is a histogram of Web-object popularity. The y-axis indicates the number of objects (using log scale) and the x-axis indicates the number of hosts that share an object (using a bin size of 25). There is a line for each of the five traces. A point on a line indicates the number of objects that are requested by the specified number of hosts. For example, the graph shows that, in the UU trace, roughly 10 objects were requested by between 126 and 150 hosts.

Figure 12 shows three important features. First, most objects are accessed by a small number of hosts; the log scale of the y-axis somewhat hides this feature. Second, the distributions appear tail heavy, as has been observed by Cunha et al. [4]. For example, at around 150 to 200 hosts, the number of shared objects has dropped considerably; after that, however, the decline from 200 to 800 hosts is much more gradual. In fact, the line for DEC continues out to 4000 hosts and varies between zero and ten objects all the way out. Third, the object-popularity pattern for all traces is similar, though traces with higher reference counts have more widely shared objects, as expected.

Figure 13 graphs the normalized request rate for objects as a function of the number of hosts that share them. Notice that every object is summarized at the same x-axis point in both Figure 12 and 13. In Figure 13, however, the y-axis indicates the average per-host per-object request rate for objects with the specified degree of sharing. The important thing to notice about this graph is that a host's per-object request rate is mostly independent of an object's popularity, though very popular objects are requested at a higher rate; this is also true for DEC, which starts to trend upward toward six at around 2200 hosts (not shown in Figure 13).

Sharing Between Clients from Different Traces

To further understand the distribution of requests to shared objects, we conducted a series of experiments in which we looked for sharing patterns among clients from different traces. These comparisons are interesting because each trace represents a totally distinct collection of users. An object shared by users from multiple traces might be considered to be of *global* interest to Internet users in general. In contrast, objects shared only within a given trace are of only *local* interest to the particular user community. In addition, we mentioned above that some degree of false shar-

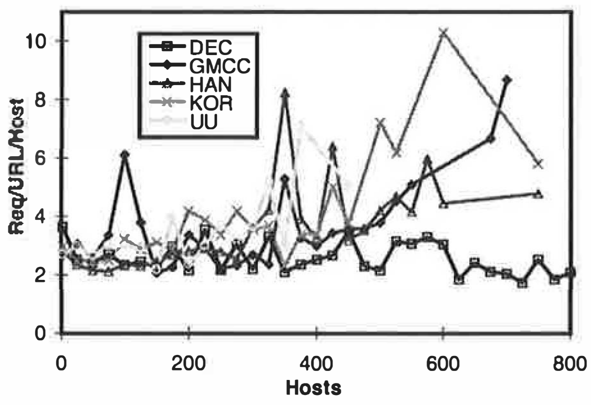


Figure 13: Graph showing the per-URL per-host request rate for objects based popularity of URL (from Figure 12).

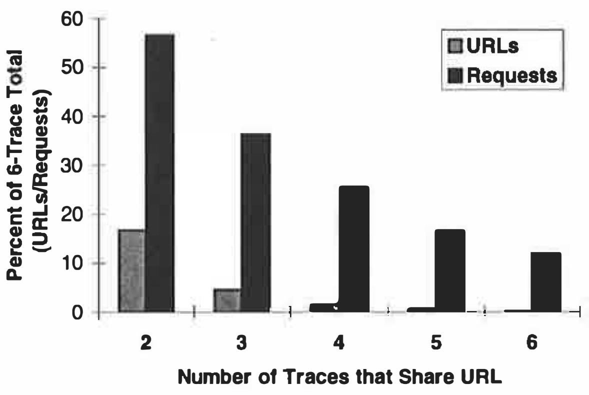


Figure 14: Inter-trace sharing among HAN, KOR, GMCC, AU, UU, and NLANR. Shows percent of shared URLs and requests for those URLs for sharing between a given number of six traces.

ing occurs within a trace, because some users use multiple client hosts. False sharing, however, is eliminated when considering sharing among multiple traces.

Figure 14 compares six traces: HAN, KOR, GMCC, AU, UU, and NLANR; DEC is excluded because its trace stores URLs using a hash code that can not be compared with the URL strings in other traces. There are five pairs of bars on the x-axis; of each pair, one bar shows shared objects and the other shows shared requests. Each pair of bars shows the amount of sharing that exists among the specified number of traces. For example, 18% of the total objects requested by the six traces are requested in at least two of the traces and 56% of total requests ask for one of these objects.

From Figure 14 we see that, as in Figure 10, the portion of requests that ask for shared objects is much larger than the portion of objects that are shared. Furthermore, we see that this gap widens as we look at sharing across more of the traces. For example, we see that only 0.2% of objects are shared by all six traces, but 16% of all of the requests ask for one of these objects. A second important observation is that a surprisingly large number of requests (16%) ask for objects that are globally shared among all six traces; recall, however, that not all of these requests will be cache hits.

Finally, we examine the nature of narrow and wide sharing. Figure 15 compares the inter-trace sharing for objects that are *narrowly* shared in one of the traces (i.e., requested by two to nine client hosts in that trace) and those that are *widely* shared in one of the traces (i.e., requested by at least ten clients); we also show objects that are *only narrowly* shared (i.e., narrowly shared in every trace in which they appear). This figure compares only four traces: HAN, KOR, GMCC, and UU; AU and NLANR are excluded because we can not distinguish sharing from locality, as described above. There are four sets of bars, each set with two bars for narrowly-shared objects, two bars for only-narrowly-shared objects, and two bars for widely-shared objects. As before, objects are counted only if they are requested from the number of traces specified under the bar on the x-axis (notice that this graph starts with one, while Figure 14 starts with two). An object that is narrowly shared in one trace and widely-shared in another trace counts as being both narrowly and widely shared, but not as only-narrowly shared.

Figure 15 shows that Web sharing tends to be bimodal. First, notice that the one-trace bars on the far left of the figure show that sharing requests are divided almost evenly between narrowly- and widely-shared objects, while there are many more narrowly-shared objects than widely-shared objects. Furthermore, the other sets of bars show that a significant portion of widely-shared objects are also globally shared, while narrowly shared objects are almost exclusively locally shared. For example, for sharing among all four traces, only-narrow-sharing requests drop to 0.2% while wide-sharing requests remain relatively high at 9%; note that 4% of requests asked for objects that were both narrowly and widely shared (i.e., narrowly shared). We thus conclude that Web sharing tends to be divided roughly evenly between objects that are shared narrowly and locally and those that are shared widely, and that many widely-shared objects are also shared globally.

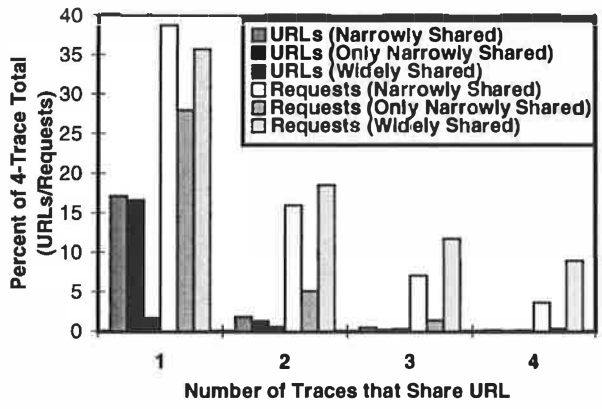


Figure 15: Inter-trace sharing among HAN, KOR, GMCC, and UU. Divides sharing into Narrow sharing, objects shared by less than ten distinct hosts, and Wide sharing, objects shared by at least ten hosts.

Summary

Our analysis presented in this section shows several key characteristics of Web client sharing patterns.

- Sharing is high (20% to 38%) and dominates single-client locality as the primary factor that determines hit rate.
- Sharing increases as the number of clients, and thus request rate, increases, while single-client locality does not increase.
- Up to 71% of requests are to shared objects, though roughly half are misses due to first-time accesses, consistency, and capacity misses. Only 15% to 28% of objects are shared.
- Most shared objects are accessed by only a few clients, though the distribution of object popularity appears to be tail heavy.
- Sharing is bimodal. Half of a trace's sharing is local to the trace and involves only a few hosts, the rest is more global, overlapping with other traces, and involves many hosts.

3.5 Implications for Cache Design

Our analysis shows that high hit rates depend on caches having sufficient clients to generate a high request rate. For example, a one-thousand client cache with 70,000 requests/day had a hit rate no higher than 28%, while a two-thousand client cache with 250,000 requests/day achieved a 45% hit rate. Furthermore, the NLANR third-level cache, whose requests are lower-level cache misses, had a hit rate of 19%. These two observations suggest that a client connected to a three-level hierarchy such as NLANR might see hits rates as high as 55% (i.e., the best hit rate of the second-level caches plus an additional 19% of that cache's misses).

The fact that sharing and hit rate increase with request rate might seem to argue for a monolithic cache structure consisting of a single-level cache designed to handle thousands or tens of thousands of clients. Latency and scalability concerns, however, argue for a more hierarchical structure.

A hierarchical structure allows caches to be placed closer to clients than does a monolithic structure, because the monolithic cache must be far enough away from clients to include the entire client pool. We have shown, however, that a relatively small group of one-thousand clients generates substantial sharing, with hit rates in the range of 25%. A more distant monolithic cache would increase request latency for the quarter of requests that could have been satisfied more locally.

A hierarchical structure also aids cache scalability. A substantial request rate is needed to achieve the 55% hit rate that our analysis indicates may be possible. For example, Table 1 shows that DEC has an average request rate of 12 requests/s. We have also computed the request rate as a function of time for every trace. This data shows that DEC has a peak request rate of 78 request/s and a peak miss rate of 55 misses/s; miss rates is important because it determines the impact a cache has on its parent in the cache hierarchy. For comparison, the proxy-cache performance study by Maltzahn and Richardson shows that peak per-processor throughput of the Squid v1.1 proxy is less than 35 request/s (when running on a Digital AlphaStation 250 4/266).[1]

We thus conclude that a hierarchical structure is the best solution for providing low-latency hits for local sharing, while achieving the substantial hit rates that are only achievable when thousands of clients share a cache.

4 Limitations of Trace Data

During the course of our study, we identified four limitations of the trace data we analyzed. We outline these limitations here in the hope of influencing proxy providers to remove these limitations from future versions of their servers.

[1]The study shows per-request CPU utilization of the CERN and Squid v1.1 servers at 15-million cycles and 7.5 million cycles respectively.

1. Including the last-modified and expiry time returned by servers would have allowed us to simulate larger caches. The lack of this information in all but the DEC trace limited simulated cache size to be no greater than actual cache size.

2. Preserving a single unique client name over time (appropriately disguised to protect user privacy) is necessary for any analysis of sharing. If a client's disguised name changes from day to day, sharing cannot be distinguished from multi-day client locality.

3. Identifying users (again appropriately disguised) in addition to their host IP address would eliminate the false-sharing problems that occur with multi-user hosts and with users that use multiple hosts.

4. The ability to compare URLs from different traces is needed in order to measure inter-trace sharing. If a hash function is used to store URLs in a more compress form, the same function should be used by all proxies.

5. Including response codes returned by servers is important for distinguishing error responses (e.g., object not found). Most traces do include response codes, but some do not (e.g., the traces used by Gribble et al. [11]). Fortunately, experiments we conducted show that the lack of response codes causes less than a 1% hit rate difference.

5 Related Work

Since the Web became the primary consumer of Internet bandwidth, studies of Web traffic have become common. Some early studies include analysis of Web access traces from the perspective of browsers [6, 4], proxies [10, 20, 22], and servers [16]. Arlitt et al. conducted a recent study of Web *server* workloads [2]. Our work is unique in two ways. First, we examine many more requests, much larger caches, and much higher request rates; we also include data from many more sites. Second, unlike the earlier studies, we use a cache simulator to examine dynamic workload characteristics such as request rates and sharing.

More recent research has used simulators to vary individual parameters on large traces. Gribble and Brewer [11] simulate a trace with 20 million requests. They show a hit rate of 56% for a 6 GB cache. By comparison, our DEC and HAN traces see a 37% and 42% hit rate, respectively, for a 6 GB cache. Gadde et al. [9, 18] evaluate their proposed directory-based cooperative proxy cache using simulation of twenty-five days of the DEC trace. They see sharing hit rates of 45% for an 8 GB cache compared to our 38%, because their simulator does not model cache coherence.

Others have analyzed Web proxy traces for different purposes. In [15], Malzahn et al. compared the performance of the two most popular Web proxy servers (CERN [14] and Squid [5]). They show how the CPU, memory, and disk utilization of the proxy servers scales with increasing request rate.

6 Conclusions

Client-side caching is a key solution for improving Web client performance and for scaling the Internet and Web servers to meet ever increasing demands. The design and assessment of cache designs can benefit greatly from a detailed understanding of Web client access characteristics. This paper characterizes Web-client access, based on an analysis of proxy cache traces containing a total of 47 million requests from 23,700 clients at seven different locations (including one third-level cache).

Our analysis shows that cache hit rates for second-level caches vary from 24% to 45%. Sharing accounts for 85% of these hits and sharing increases with request rate. The hit rate of the third-level cache we examined was lower at 19%, because lower-level caches filter locality and lower-level sharing from its request stream.

Desired cache size varies between 2 and 10 GB. Small client populations need 1-GB of cache per 35,000 requests/day and larger populations 1-GB per 70,000 to 100,000 requests/day, though artificially removing clients from a population does not cause a proportional reduction in cache size.

Using the Squid v1.1 and CERN cache coherence protocol, between 2% and 7% of all requests are consistency misses to unmodified objects; that is these requests were hits on expired objects that had not actually changed at the Web server.

Requests to shared objects account for 71% of total requests, but only 24% of requested objects are shared. Most of these shared objects are accessed by only a few clients, though object popularity appears to be tail heavy and a few objects are accessed by most clients. Shared requests exhibit bimodality based on an even division of requests to objects shared narrowly by a few clients and objects shared widely by many clients. Unlike narrow sharing, wide sharing tends to be global. 6% of the total 11.7-million requests in HAN, KOR, GMCC, and UU, for example, ask for objects shared by all four traces.

Finally, our results argue for a cache hierarchy whose first level is close to clients; a one-thousand client cache should have hit rates of around 25%. One or more higher levels of caching are needed to expose the additional sharing present only in larger client populations (i.e., populations of a few thousand clients or more). For large populations, we have observed hit rates of 45% and, for the entire hierarchy, hit rates of 55% seem achievable.

Acknowledgments

We would like to thank everyone who provided us with proxy traces, particularly: Christain Grimm at the University of Hanover, Germany; Jaeyeon Jung with the Nation Wide Caching Project of Korea; Tom Kroeger, Jeff Mogul, and Carlos Maltzahm for making the DEC logs available; Tim Crisall at Grant MacEwan Community College, Alberta; Henny Bekker at Utrecht University, the Netherlands; Duane Wessels at NLANR; Maciej Kozinski at Nicolas Copernicus University, Poland; and our anonymous university. Thanks also to Norm Hutchinson and our shepard, Richard Golding, for their comments on earlier versions of this paper. Finally, thanks to the folks in the UBC system's lab for putting up with our long-running data-intensive simulations.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ATM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS'96*, May 1996.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirrif, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[4] A. Bestavros C. R. Cunha and M. E. Crovella. Characteristics of www client-based traces. Technical report, Boston University, Jul 1995.

[5] Squid Internet Object Cache.
URL: http://squid.nlanr.net.

[6] L. D. Catledge and J. E. Pitkow. Characterizing browsing strageqies in the World-Wide Web. In *Proceedings of the Third WWW Conference*, 1994.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *USENIX 1996 Annual Technical Conference*, January 1996.

[8] National Laboratory for Advanced Network Research (NLANR) Proxy Traces.
URL: ftp://ircache.nlanr.net/Traces/.

[9] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large Internet caches. In *Sixth Workshop on Hot Topics in Operating Systems*, 1996.

[10] Steven Glassman. A caching relay for the world wide web. In *Proceedings of the First Interntional Conference on the WWW*, 1994.

[11] S. D. Gribble and E. A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems '97*, 1997.

[12] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[13] B. Krishnamurthy and C. E. Wills. Study of piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems '97*, 1997.

[14] A. Luotonen, H. Frystyk, and T. Berners-Lee. W3C httpd. URL: http://www.w3.org/hypertext/WWW/Daemon/.

[15] C. Maltzahn and K. J. Richardson. Performance issues of enterprise level web proxies. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997.

[16] J. Pitkow and M. Recker. A simple yet robust caching algorithm based on dynamic access patterns. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, 1994.

[17] J. Lorch R. Malpani and D. Berger. Making world wide web caching servers cooperate. In *Fourth International World-wide Web Conference*, pages 107–110, Dec 1995.

[18] J. Chase S. Gadde and M. Rabinovich. Directory structures for scalable internet caches. Technical Report CS-1997-18, Duke University, 1997.

[19] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, Oct 1996.

[20] Jeff Sedayao. "mosaic will kill my network!" - studying network traffic patterns of mosaic use. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, 1994.

[21] Digital Equipment Corporation Proxy Traces.
URL: ftp://ftp.digital.com/pub/DEC/traces/proxy/tracelistv1.2.html.

[22] D. Wessels. Intelligent caching for world-wide web objects. Master's thesis, Washington State University, 1995.

[23] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the IEEE International Conference on Distributed Computer Systems*, pages 2–11, June 1993.

A Highly Scalable Electronic Mail Service Using Open Systems

Nick Christenson, Tim Bosserman, David Beckemeyer
EarthLink Network, Inc.
Information Technology
Pasadena, CA 91107
npc@earthlink.net, tboss@earthlink.net, david@earthlink.net

Abstract

Email is one of the most important of the Internet services. As a very large, fast growing, Internet Service Provider, EarthLink requires a robust and powerful email architecture that will support rapid expansion. This paper describes such an architecture, its motivations, its future, and the difficulties in implementing a service on this scale.

1 Introduction

Electronic mail has a special standing in the ranks of Internet services. Of the direct services an ISP provides to its subscribers, email is certainly one of the most important. As a consequence, it requires special attention to keep email running as well as expected. Additionally, there are several issues that are far more problematic for email than for other services. Email typically requires more resources than any other service. This is because the storage needs, the processing power, and the bandwidth requirements are extreme. Furthermore, there are problems regarding authentication and provisioning that are often not required for other services.

Despite its criticality, little work has been made available publicly on robust, large scale electronic mail systems. The few references we have found, such as [Grubb96], have neither addressed what we consider to be the key problems, nor have they been able to scale to the capacity that we require. This isn't too surprising. Providing email service for hundreds of thousands or millions of users is a problem nobody had to solve before the advent of the national or international Internet or on line service provider. Until now, none of these organizations have chosen to come forward and publish their service architecture.

Additionally, the architecting of very high performance truly distributed services is still in its infancy. The issues of distributed storage and load balancing have few, if any, available solutions that are both robust enough and perform satisfactorily for our purposes.

The astute reader will certainly notice that the architecture we describe here bears a great deal of similarity to what we have already described as our news service architecture [Christ97a]. This, of course, is no accident. We've found a general set of principles which we have adapted to meet the needs of both services, and many of the important issues discussed in that article are equally applicable here.

In the design of any of our service architectures, we have several requirements that must be met before we would consider deployment. For email, the first of these is message integrity. It is absolutely essential that messages, once they are accepted by our system, be delivered to their proper destination intact. Second, the system must be robust. That is, in as much as is possible, the system should survive component outages gracefully. Additionally, the entire system design should minimize the number of single points of failure. Third, the system must be scalable. When EarthLink began deployment of the current architecture, in January of 1996, we had about 25,000 subscribers. In September of 1997, EarthLink provided email service for over 350,000 subscribers with a 99.9+% service uptime record. In fact, we expect the current system to scale to well over 1,000,000 users without significant alteration of the architecture as presented here. Moreover, one should be able to accomplish the scaling of any service with a minimum of outage time, preferably with none. In all cases the performance of the service must be at least adequate, and the service must be maintainable. Problems must be easily recognizable, and it should be obvious, whenever possible, what is the cause of the outage. Further, its solution should be easy to implement and, in the meantime, the impact of the outage should be small and locally confined. Finally, we would like the service architecture to be cost-effective, not just in terms of equipment acquisition, but, more critically, in terms of maintenance.

2 Architecture Description

There are several logically distinct components which make up the operation of EarthLink's email service. The first, which we call the "front end" of our email system (front defined as the portion which receives data from the Internet) are the systems that receive mail for "username@earthlink.net". These machines are also called the SMTP machines. The second component is the POP service, the servers to which subscribers connect to retrieve their mail. These same computers also send the mail originating from our subscribers to the Internet. (At the time of this writing, EarthLink has not deployed an IMAP service.) The third component is the file servers, which do nothing except store the mailboxes, mail queues, and auxiliary files associated with the email service. The fourth component is the authentication database which holds the username/password information, information on where mailboxes are stored, and data on auxiliary email services to which that account may have subscribed. This architecture is demonstrated graphically in Figure 1 included at the end of this paper. All the servers we use in this architecture, except for the file servers, are running some flavor of Unix.

With the exception of our file servers and the authentication database servers, our architecture calls for all of the servers involved in our email service (as well as all our other services) to be dataless. That is, each server should store on local disk its own operating system, service software, swap space, temporary file storage for nonessential data—and nothing else. This allows us to add or subtract servers from service with which the Internet or our subscribers interact without affecting the data stored.

2.1 Front End

Mail Exchange (MX) DNS records for earthlink.net and mail.earthlink.net point, with high preference, to a series of servers in Round Robin. These servers all run a recent version of the freely distributable stock sendmail [Allman86] as their SMTP MTA (Simple Mail Transfer Protocol, Mail Transfer Agent; see [Postel82] for details). Writing and maintaining an SMTP MTA is a difficult and expensive task. Therefore, we have geared our architecture to allow us to use sendmail without any source modification. Since sendmail typically undergoes several significant revisions during each calendar year, it's important that we be able to use sendmail in a form as close to the stock distribution as possible, since updating a modified sendmail every few months to reflect local modifications would be about as time consuming as maintaining our own MTA.

An electronic mail message to be delivered to "username@earthlink.net" follows the DNS MX records for earthlink.net. We maintain several machines with high precedence MX records using Round Robin DNS to distribute the load. If any of these machines become overloaded or otherwise unavailable, we maintain a single machine with a lower precedence MX record to act as a spillway. This server does not deliver email directly, but it does hold it for forwarding to the first available front end server. Our backup MX machine could easily be configured to deliver email itself, but we have chosen not to do this to protect against the possibility of transient errors in the mailbox locking process. This way, if the locking scheme becomes overloaded, we have a server that can still accept mail on behalf of the "earthlink.net" domain. It is our intention to minimize at all times the amount of email queued around the Internet for delivery to EarthLink.

The key to using the stock sendmail in our architecture is to insure that the sendmail program itself attempts to do no authentication or lookup of user names. This is actually quite simple to do. One merely must remove the "w" flag in the entry for the local delivery agent in the `sendmail.cf` file. Even if one runs a standard authentication scheme, we've found that this modification provides a considerable performance boost if one has a large, unhashed (i.e. linear lookup) `passwd` file, and the service is not completely inundated with email intended for non-existent users.

The portion of the mail reception service that we did modify heavily was the mail delivery agent. This is the program that receives the mail from the MTA and actually appends it to the user's mailbox. On most systems, this program is `/bin/mail`. The sendmail distribution provides a delivery agent called `mail.local` which we have rewritten to use our authentication methods and understand how we store mailboxes. This is a small program which hasn't changed substantially in years, so it is easy to maintain; hence, it is a better place to add knowledge about our email architecture than a moving target like sendmail.

In addition to authentication and mailbox location, the mail delivery agent also knows about mailbox quotas which we impose on our subscribers. If the current mailbox size is over the quota for that user, the default being 10 MB, then the message is bounced back to the MTA with reason, "User npc, mailbox full." In addition to preventing resource abuse on the part of subscribers, this also helps mitigate possible damaging effects of mail bombing by malicious people on the Internet. We believe that a 10 MB quota is quite generous, especially considering over a 28.8 modem using very high quality line speeds and no network bottlenecks, one could expect to take over an hour to download the contents of a 10 MB mailbox.

2.2 POP Daemon

What we call the "back end" of our architecture is a set of machines using Round Robin DNS which act as the POP servers. They are the targets of the A records, but not the MX records, for earthlink.net and mail.earthlink.net. These are the servers to which the subscribers connect to retrieve and send email. If these machines receive email bound for user@earthlink.net (from our subscribers, Internet machines compliant with the SMTP protocol must follow the MX records and send this message to a front end machine), these messages are redirected to our SMTP machines at the front end. The POP servers do no local delivery. They do, however, deliver directly to the Internet. We've debated the notion of having the POP servers forward mail on to yet another set of servers for Internet delivery, but have thus far elected against it. The benefit to doing this would be further compartmentalization of physical server function by logical operation, which we consider to be inherently good. We'd also reduce the likelihood of POP session slowdowns in the case that a subscriber floods the server with an inordinate amount of mail to be delivered. If the POP and Internet delivery functions were separate, the POP server would expend very few resources to hand this mail off to the delivery servers, whereas it would otherwise be required to try to send and possibly queue this mail itself. On the other hand, we don't observe this being a significant problem. Additionally, if we had a separate Internet delivery service within our mail architecture, we'd have to deploy an additional machine to maintain our complete N+1 redundancy to every component, at additional cost. Someday, we probably will make such a separation, but it does not seem to be justified for the present volume.

Like the delivery agent, the POP daemon must also know about both our modified authentication system and mailbox locations. The base implementation we started with was Qualcomm's POP daemon version 2.2, although like `mail.local`, we have modified it substantially. In the near future, we plan to completely rewrite the POP daemon tuned for efficiency in our environment implementing many of the lessons learned from developments in Web server software.

2.3 Mailbox Storage

On a conventional Unix platform, mailboxes are typically stored in `/var/mail` or `/var/spool/mail`. The `passwd` file, used to store valid user names for incoming mail and to authenticate POP connections, is usually located in `/etc`, and mail which cannot be delivered immediately is stored in `/var/spool/mqueue`. This is where our mail architecture started out as well, but we made some significant changes as we went along.

As with Usenet news, we use the Network Appliance [Hitz95] family of servers as our network file storage for essentially the same reasons: Very good performance, high reliability of the systems, easy maintenance, and the advantages provided by the WAFL filesystem [Hitz94].

Due to performance considerations, the spool is split across several file servers and each is mounted on the SMTP and POP servers as `/var/mail#`, where `#` is the number of the mount point, in single digits as of this writing.

Currently, we're using version 2 of the NFS protocol. While version 3 does give some significant performance benefits, we give it all back because of the implementation of the READDIRPLUS procedure [Callag95] which prefetches attribute information on all files in that directory, whether we need them or not. Since we store a large number of files in the same directory and are only interested in operating on one of them at a time, this is significant overhead that we don't need. On balance, for our email system, the performance difference between versions 2 and 3 of the NFS protocol is so small as to defy precise measurement. We typically change it whenever we suspect the current version might be responsible for some strange behavior we notice. The NFS version has never turned out to be the problem, so we usually leave that version in place until we feel the need to change it again in order to eliminate it as a factor in some other problem we face.

Even though the Network Appliance's WAFL filesystem provides excellent protection against the performance penalties one normally encounters when there are very large numbers of files in a single directory using most other file systems, there are still significant advantages to breaking them up further. Since we have more files and require more storage and throughput than we can realize with any one file server, we need to split the spool up and provide some mechanism to locate mailboxes within this tree. So, we create a balanced hash for each mailbox over 319 possible subdirectories (the prime base of the hash) and divide these directories over the number of file servers that compose the spool. Thus, a path to a mailbox may look something like `/var/mail2/00118/npc`. The POP daemon and the local delivery agent are the only parts of the mail system that need to know about these locations.

Once we have this mechanism for multiple locations of mailboxes in place, we are able to extend this to allow us to dynamically balance the mailboxes or expand capacity. In addition to the notion of the "proper" location of each mailbox, both `mail.local` and `popper` (the POP daemon) understand the notion of the "old" mailbox location. If the system receives email for a given mailbox, `mail.local` checks the "proper" location for the mailbox, and if it finds it there, appends the message in the normal manner. If the mailbox isn't there, it checks the "old" location for the mailbox. If it is found there, `mail.local` locks the mailbox in both locations, moves it to the new location and appends the message in the normal manner. If the mailbox exists in neither place, it creates a new mailbox in the "proper" location. The POP daemon also knows this information. It looks in the "proper" location first, and if it is not there, it consults the "old" location. In either case, it operates on the mailbox entirely in the place where it was found.

Only `mail.local` actually moves the mailbox. We felt that it would be better to confine the mailbox moving logic in the simpler of the two programs. Because the mailbox can only be in one of the two places and the delivery agent and POP daemon use a common locking system (described below), there's no danger of confusion as to the mailbox location.

The data on what constitutes the "old" and "proper" mailbox locations are kept in the authentication database (explained below), and this information is returned to the client process when authentication information is accessed.

This feature has a major benefit for our mail system. This allows us to move large numbers of mailboxes around without interrupting service. For example, if we have three file servers containing mailboxes and they are either getting to be full or running out of bandwidth capacity, we can create a new mount point, `/var/mail4` for instance, mount a new file server on the mail servers, create the hash value subdirectories that will reside there, and then slide a new `mail.local` and `popper` in place (POP daemon first!) that know which of the subdirectories from each of the first three file servers will now be housed on the fourth. Then, as mailboxes receive new mail, they are moved onto the new file server. After a few hours, days, or weeks (depending on how much of a hurry we're in), we can start a second process of individually locking and moving mailboxes independent of any other activity on the systems. Thus, we have now expanded our email system without any downtime.

2.4 Authentication

One thing we quickly realized is that the standard Unix authentication systems were wholly inadequate for a service of this magnitude. The first problem one runs into is that depending on the specific operating system, one is typically confined to between 30,000 and 65,535 distinct user identities. Fortunately, since none of these users have shell access to these servers (or any access other than POP access), we can have a single UID own every one of these mailboxes as long as the POP daemon is careful not to grant access to other mailboxes without re-authenticating.

While this postpones several problems, it isn't sufficient by itself to scale as far as we'd like. First, several Unix operating systems behave quite strangely, not to mention inappropriately, when the `passwd` file exceeds 60,000 lines. This isn't completely unexpected—after all how many OS vendor test suites are likely to include these cases—but some of these problems manifest themselves a great distance from the `passwd` file and, thus, can be difficult to track. Just as important, when the `passwd` file gets this large, the linear lookups of individual user names become expensive and time consuming. Therefore, the first thing we did was make a hashed `passwd`-like file using the Berkeley NEWDB scheme [Seltze91] that both `popper` and `mail.local` would consult for authentication. This eliminated the need to carry a large `passwd` file and greatly increased performance of the system. A separate machine working in a tight loop continually rebuilt the hashed `passwd` file as the text file was continually being modified by the the new account provisioning system.

The next logical extension of this was to store the `passwd` file in a SQL database and replace `getpwnam()` calls with SQL equivalents. This provides another quantum improvement. First, this eliminates the necessity of continually rebuilding the hash file from the flat file, with savings in processor and delay times for user account modification. Second, this database may be used by other applications including RADIUS [Rigney97], Usenet news, etc.... Third, it's a logical place to store additional important information about that account. For example, when a username lookup by `mail.local` or a username/password pair is authenticated for `popper`, the "old" and "proper" mailbox locations are returned to the application rather than having to be stored in flat files on the system or hardcoded into the respective binary. We also intend to use this database as a repository for a great deal more information, for example storing variable mailbox quotas and lists of domains from whom to refuse mail on a user by user basis, etc....

Obviously, this database is critical to not only the operation of our electronic mail system, but to other components of our overall service architecture as well. If the authentication service isn't operating, electronic mail comes to a halt. Because of this, we have taken special pains to make certain that this application is always on line by using a clustered system with failover using a dual attached storage unit for the database to meet our high availability requirements. If it becomes necessary, we can still fail over to the old common hashed `passwd` file with only a marginal loss of functionality and performance.

2.5 File Locking

In any distributed system, concurrency issues are of paramount importance. In our email system, these manifest themselves in terms of file locking. It is so important, we have given the topic its own section in this paper to discuss the issues which the implementor faces.

Yesterday

For data stored on local disk, `flock()` suffices to ward against two processes attempting to process the same message or modify the same mailbox at the same time. Since all of our persistent data is accessed over NFS, this presents some significant problems for us.

When using `flock()` on an NFS mounted file system, these calls get translated to requests via `rpc.lockd`. Now, lockd isn't the most robust file locking mechanism ever devised. It isn't advisable to bank on lockd working entirely as advertised. In addition to this, many systems have lockd tables too small for our purposes. We can routinely require thousands of outstanding lock requests on a given NFS mounted filesystem at any one time, and few commercial solutions have lock tables large enough and/or lock table lookup algorithms fast enough to meet our needs.

Today

Therefore, wherever possible, we use the file system to perform locking. This consists of requesting an `open()` system call to create a new (lock) file with the `O_EXCL` flag of a file of a predetermined name, typically mailboxname.lock, in a given location, which would typically be the mail spool. In our case, in order to keep the spool directory sizes down as much as possible and performance as high as possible, we store these files on their own shared file system.

This may set off alarms in the heads of those familiar with NFS. One might well ask, "How can you be certain that this is atomic on an NFS system? How do other clients know that one has locked a given file?" Recent implementations of NFS client software ignores the attribute cache on `open()` calls which attempt to exclusively create a new file. Note, however, that other `open()` calls do not ignore the attribute cache. This means that if a process's exclusive `open()` on the lock file succeeds, that process has successfully locked that file. This allows us to use file locking on the mailboxes, as long as we are mortally certain that all NFS clients operate in exactly the same way. One can find both NFS v2 and v3 implementations that behave this way. It cannot be overstated how important it is to be certain that all NFS clients behave in this manner.

It is always possible that the process which creates the lock will die without having the opportunity to remove it. For this reason, all processes creating locks must touch the lock files to update their attributes periodically so that if these processes die, after a certain amount of time other processes will know that an existing lock is no longer valid and can be eliminated. Therefore, we need a function that, again, will bypass the cache and be guaranteed to immediately update the attributes on the lock file.

Let us suppose that one process on one NFS client has created a lock on the mailbox "npc". Let us also suppose that a process on a different NFS client then tried to lock that mailbox immediately afterwards and discovered the existing lock, as it must. It's always possible that the first process has somehow died, so it's important to understand how long the second process must wait before it can assume that the first process no longer exists, at which time it can delete the lock file, lock the file itself, and perform operations on that mailbox. Again, let us suppose that all the NFS clients are set to refresh their mailbox lock every five minutes, and that the NFS attribute cache is set on each client to be three minutes.

One scenario is for a process on client1 to successfully lock the mailbox and then have client2 immediately attempt to lock the same mailbox and fail. At this point, the information on the locked mailbox is saved for three minutes, the duration of the attribute cache, after which client2 gets the same attribute information as before, because five minutes has not elapsed, therefore client1 has not yet refreshed the lock. At the five minute mark, client1 refreshes the lock file using `utime()`, since it also bypasses the NFS cache and operates synchronously on the lock file, but client2 has not noticed because it will be looking at the attributes in its cache until the six minute mark, when its cache expires, and it can now gets the updated information. This is represented graphically in Figure 2.

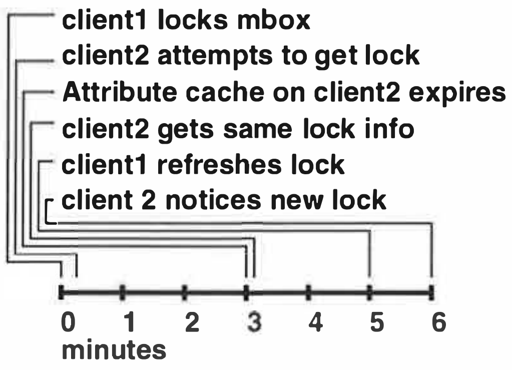

client1 locks mbox
client2 attempts to get lock
Attribute cache on client2 expires
client2 gets same lock info
client1 refreshes lock
client 2 notices new lock
0 1 2 3 4 5 6
minutes

Figure 2

The worst case scenario is presented in Figure 3. Here we have client1 creating a lock on a mailbox and then immediately dying. Just before the lock is scheduled to be updated, client2 attempts to lock the mailbox and fails. Client2 cannot learn that the lock hasn't been updated until just before the eight minute mark, at which point it has license to remove the lock file and proceed with its actions.

Unfortunately, this potentially gives us a window of eight minutes in which real users may not be able to access their mailboxes under pathological conditions. For example, if the subscriber interrupts a POP session at the wrong moment, the POP daemon on the mail server may exit without cleaning up its lock file. Further, we explain below why we must delay even longer than this to allow for other concerns.

If the file servers ever get saturated with requests, the server can seem to "disappear" to client processes for many seconds or even minutes. This can happen as part of normal subscriber growth if one does not upgrade the capacity to handle load before it is needed. In these circumstances, problems usually manifest themselves as a sudden change in performance response from acceptable to unacceptable over a very small change in load. The mathematicians would call this a catastrophic response, where "...gradually changing forces produce sudden effects." [Zeeman77] If a file server's load is near, but not at the critical point, it can be pushed over the edge by a sudden change in the profile of normal email use or by a small number of malicious or negligent individuals.

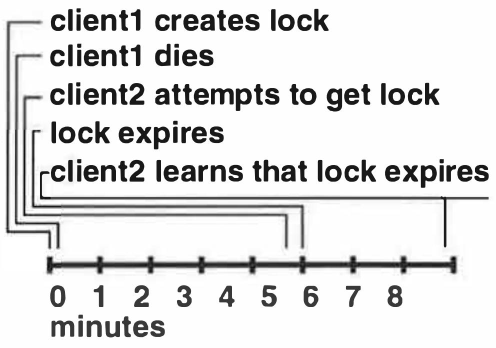


Figure 3

It is self-evident that one wants to provide enough surplus performance to prevent small changes from breaking the performance envelope, and certainly we strive for this, yet it is not always possible. As an example, consider a two week period in August 1996 where the total volume of email EarthLink was called upon to handle doubled for reasons that are still not fully understood. While not routine, these events are not uncommon in the ISP business and, because the subscriber has a much greater ability to impact service, represent a fundamental difference between providing Internet services and, for example, providing electric power or dial tone service.

In any case, when one enters into one of these catastrophic regimes, one often encounters pathologic behavior on the part of any or all of the components of the service. Client requests can come too fast for the server(s) to handle; consequently the RPC packets can get dropped before they are processed. This can result in retransmissions by multiple clients, and on top of an already saturated system, the problem is compounded.

Let us suppose that we have a saturated system where the client base demand is 105% of the server's capacity to deliver it over a given period of time, not counting the load put on the server because of retransmissions. Each of these clients will now retransmit their requests after a number of tenths of seconds specified by the `timeo` value in the `/etc/vfstab` or equivalent file. If this request does not receive a response, the client waits for twice `timeo` and retransmits again. This process is repeated until the value of the `retry` variable is reached. If `retry` is exceeded, then the client prints a message, typically "NFS server raid not responding, still trying," and continues to retry at the maximum retry value. This maximum value will never exceed 30 seconds [Stern91]. Under these conditions, we cannot achieve a "steady state" condition, the amount of traffic grows, quite dramatically, without bound until something breaks.

If this condition were to persist for 30 minutes, at this time as much as 25% of the requests sent to the servers may be over 5 minutes old. Note that this represents a true pathological condition, it's highly unlikely that a client machine would either be able to maintain this load given the lack of responsiveness of the server, or that the client load would be constant, but we haven't yet developed our mathematical models sufficiently to account for all the known variables, so we're being conservative. Given these assumptions, if we are adding $2,000,000$ new email messages to our spool in a day, a half an hour of operation at this level of saturation with a lock timeout of only 5 minutes, we must expect there to be on the order of a thousand mailbox corruptions due to multiple processes proceeding to modify mailboxes on the assumption that they have exclusive access to it. This is because they have encountered expired lock files which are actually valid, the owning process just hasn't been able to get the server to ack it's update of the lock file. The mathematics behind this analysis and an in depth examination of the ramifications of this will be fully explored in [Christ97b]. Therefore, it is important that our locking mechanism allow for the possibility that a client process may not be able to get their request through to the server for several minutes after the normal locking timeout window has closed. We use a lock timeout value of 15 minutes to allow for this possibility.

With regard to locking, one area of concern we have is with sendmail. Current versions want to use `flock()` to lock files in mail queues. On our email system, the depth of these queues is extreme and the number of processes that can concurrently be trying to drain them can be as high as several hundred per machine, requiring a large number of outstanding lock requests at any one time, often too many for either the client lock daemon or the file server to accommodate. Because of this, we have two choices. Either we can put the mail queues on locally attached disk, violating our stateless architecture principle and losing the benefits of the WAFL file system in handling directories with large numbers of files, or we can modify sendmail to use a different locking mechanism, thus violating our intention to use an unmodified SMTP MTA. Fortunately, the current sendmail implementation has very modular locking code which can be easily replaced without fundamentally altering the distribution. However, we'd like any folks working on sendmail to consider allowing a preference for various locking mechanisms to be #define'd in the source code.

Tomorrow

While the mailbox locking mechanism we've just described has worked satisfactorily, it is not without its drawbacks. One drawback is the fact that locks may be orphaned, and other clients must wait up to 15 minutes before being able to assume they are no longer valid. Another drawback is that the synchronous NFS operations we employ greatly increase the load placed on the NFS servers which hold the lock files.

Therefore, we are in the process of designing and building our next generation lock management system. In accordance with our design parameters, what we really want is a distributed lock system with no single points of failure. It has to maintain state in the case of a crash or hardware failure, and it must be able to handle at least several hundred transactions per second.

We tried using a SQL database for this purpose, but we were not satisfied with the performance. A program like a large commercial database such as this requires too much overhead to be efficient in this manner. However, we can learn from the database style locking mechanisms and, essentially, strip away those portions of the database system which we don't need to create our own lean and mean network lock server.

We plan to deploy two machines clustered together around a shared RAID system to act as our lock service. If the primary machine were to suffer some form of failure, the other would take over with a target transition time of less than five seconds. We intend to deploy the same hardware configuration that we use for our authentication database. All the lock requests get written to the file system using unbuffered writes before they are acknowledged so that in case of machine failure there is no loss of state.

The clients open up a socket to the lock daemon on the lock server and request a lock for a given mailbox, which the daemon either accepts or denies. If it is denied, the client waits for some pseudo–random time and tries again. We project that this system will scale well into the millions of mailboxes for a single set of lock managers. To get this scheme to scale indefinitely, it's a simple matter of having the clients query different lock servers for different ranges of mailbox names.

3 Operation

One of our primary design goals was to deploy a system that would be cost–effective to maintain. This service accomplishes those goals in several ways.

First, by centralizing authentication in a single system, we reduce the problems associated with both maintaining multiple parallel authentication systems and insuring that they are synchronized. This is a considerable overhead savings.

Second, one of the key criteria in selecting the Network Appliance as our storage system was its ease of maintenance. Because its operating system has been stripped down, eliminating functionality not necessary to its operation as a file server, the server is less likely to fail and, if it does fail, it is easier to discover and remedy the problem due to the greatly reduced number of degrees of freedom presented by the operating system.

Third, because the POP servers themselves are dataless, they require much less maintenance than their stateful equivalents. The only files which differ between these computers are those that contain their host names and/or IP addresses. This means that new servers can be brought online in a very short time via cloning an active server. Just as significant, it means that since these machines contain no important persistent data (aside from the operating system), there are few reasons for system administration to log on to the system and make changes. This helps eliminate one of the arch–nemeses of distributed computing—"state drift," the tendency for systems intended to be identical or nearly identical to become more and more different over time.

At EarthLink, one of the things we do most often is to grow an existing service to accommodate more subscribers. The efforts we have made to allow this to happen easily and with a minimum of interruption contribute greatly to lowering the cost of operation. We've already explained how we use the concept of "old" and "proper" mailbox locations to scale both file system storage and bandwidth by adding additional file servers easily and with no downtime. The network implementation we're using at this time is switched FDDI, which also scales well. As we've already mentioned the POP servers are dataless and, therefore, should lack of these resources present a problem, in very little time, and again, with a minimum of effort, we can clone and deploy a new system. This results in our email service being extremely scalable on short notice.

We attempt to maintain $N+1$ redundancy in every possible component of the system. Our data storage systems use RAID to protect against single disk failure. We keep extra data storage servers near–line in case of failure for rapid exchange with the downed system. We keep extra FDDI cards in the switch and an extra switch chassis nearby in case these components fail. We also keep one more SMTP and POP server online than loading metrics indicate is necessary. Thus, if one fails, we can pull it out of Round Robin DNS without impacting service, aside from the problems caused by the initial component failure. Additionally, we get the benefit of not having to repair the failed server immediately. Instead, we can take time to ensure that everything else is in proper running order, and then we can diagnose and repair the failed server at our leisure. On top of all this, we use a monitoring system that flags problems with each component of the service in our Network Operations Center, which is staffed 24x7x365 and contacts appropriate on–site personnel.

4 Shortcomings

We consider the architecture presented above to have considerable merit as one of the better solutions available for satisfying high volume mail service. It is, of course, not without its limitations, some of which we mention here. One of the first problems is with sendmail as an MTA. When Eric Allman developed the original sendmail, it was not envisioned that it would still be in service over fifteen years later and be pushed, rewritten, and extended to the extent that it has. It is a testament to the skill of its creator and maintainer that it has performed as well as it has for this long. Nonetheless, if one were to code an SMTP MTA today, we doubt anyone would want it to take the form of sendmail. Despite this, we don't see an MTA that would provide enough significant advantages that we would want to migrate to it in the immediate future. Of course, these statements about sendmail could have been uttered five years ago without alteration. The bottom line is that we would prefer to run an SMTP MTA that is tighter, more efficient, and has fewer potential places for security bugs to creep in, but there isn't one available that meets our needs at this time.

Probably the biggest problem with our architecture is that due to the nature of NFS, when we add additional file servers to address our performance and storage needs, we end up adding multiple single points of failure. Despite the fact that the Network Appliance file servers have been quite stable and recover quickly from problems, we feel that this is not easily scalable forever. Therefore, it is our opinion that at some point we need to abandon NFS as our distributed systems protocol for something better.

Our ideal protocol would be very high performance; be completely distributed and, thus, highly scalable, local failures would cause local, not global outages, and would allow for redundant storage that eliminates local single points of failure. Unfortunately, given the current state of distributed computing, it's hard enough to find a system that adequately addresses one of these points, and nothing seems close to providing good solutions for all of them. Consequently, we are currently in the process of designing our own distributed system to accommodate our next generation architecture requirements.

5 Current Data

Today, the system described here is in operation as EarthLink's core electronic mail system. At the time of this writing, this system supports about 460,000 mailboxes for over 350,000 users. The system processes, incoming and outgoing, over 13,000,000 email messages each week. This means we average about 20 incoming messages each second. We average about 20 new POP connections/second and hold open about 600 concurrent active POP daemons at peak time, with spikes to over 1000 concurrent outstanding POP connections at any one time.

6 Conclusion

In conclusion, we believe we have architected a mechanism to extend a standard, freely distributable, open systems email system to handle from hundreds of thousands to millions of distinct email accounts with a minimum of modification to the underlying components. We also believe that this system meets, to the best of our ability to deliver, the required criteria we set out in the Introduction.

7 Acknowledgments

The authors of this paper are by no means the only folks who have put a lot of effort into the development and operation of this system. We wish to especially thank Jay Cai and Max Chern who did a significant portion of the software development on this system. Thanks to Steve Dougherty and Mark Wagnon, who provided helpful comments, and to Jim Larson, who provided valuable input on the precise mathematics of packet retransmissions. Also, a great deal of thanks go to Trent Baker and his system administration team who maintain all our services: Gianni Carlo Contardo, Jason Savoy, Marty Greer, Alwin Sun, Horst Simon, Jewel Clark, Tom Hsieh, Lori Barfield, David Van Sandt, Larry Wang, Hong Zhang, and Kenny Chen. We also wish to extend a special thank–you to Scott Holstad who made many excellent editorial improvements to early versions of this paper.

References

[Albitz97] P. Albitz, C Liu, *DNS and BIND, 2nd Ed.*, O'Reilly & Associates, Inc., Sebastopol, CA, 1997, p. 212.

[Allman86] E. Allman, Sendmail: An Internetwork Mail Router, *BSD UNIX Documentation Set*, University of California, Berkeley, CA, 1986.

[Callag95] B. Callaghan, B. Pawlowski, P. Staubach, *RFC 1813, NFS Version 3 Protocol Specification*, June 1995.

[Christ97a] N. Christenson, D. Beckemeyer, T. Baker, A Scalable News Architecture on a Single Spool, *;login:* vol. 22 (1997), no. 3, pp. 41–45.

[Christ97b] N. Christenson, J. Larson, Work in progress.

[Grubb96] M. Grubb, How to Get There From Here: Scaling the Enterprise–Wide Mail Infrastructure, *Proceedings of the Tenth USENIX Systems Administration Conference (LISA '96)*, Chicago, IL, 1996, pp. 131–138.

[Hitz94] D. Hitz, J. Lau, M. Malcom, File System Design for an NFS File Server Appliance, *Proceedings of the 1994 Winter USENIX*, San Francisco, CA, 1994, pp. 235–246.

[Hitz95] D. Hitz, An NFS File Server Appliance, `http://www.netapp.com/technology/level3/3001.html`.

[Postel82] J. Postel, *RFC 821, Simple Mail Transfer Protocol*, August 1982.

[Rigney97] C. Rigney, A. Rubens, W. Simpson, S. Willens, *RFC 2058, Remote Authentication Dial In User Service (RADIUS)*, January 1997.

[Seltze91] M. Seltzer, O. Yigit, A New Hashing Package for UNIX, *Proceedings of the 1991 Winter USENIX*, Dallas, TX, 1991.

[Stern91] H. Stern, *Managing NFS and NIS*, O'Reilly & Associates, Inc., Sebastopol, CA, 1991, chapter 12.

[Zeeman77] E. Zeeman, *Catastrophe Theory, Selected Papers 1972-1977*, Addison–Wesley Publishing Company, Inc., Reading, MA, 1977, p. ix.

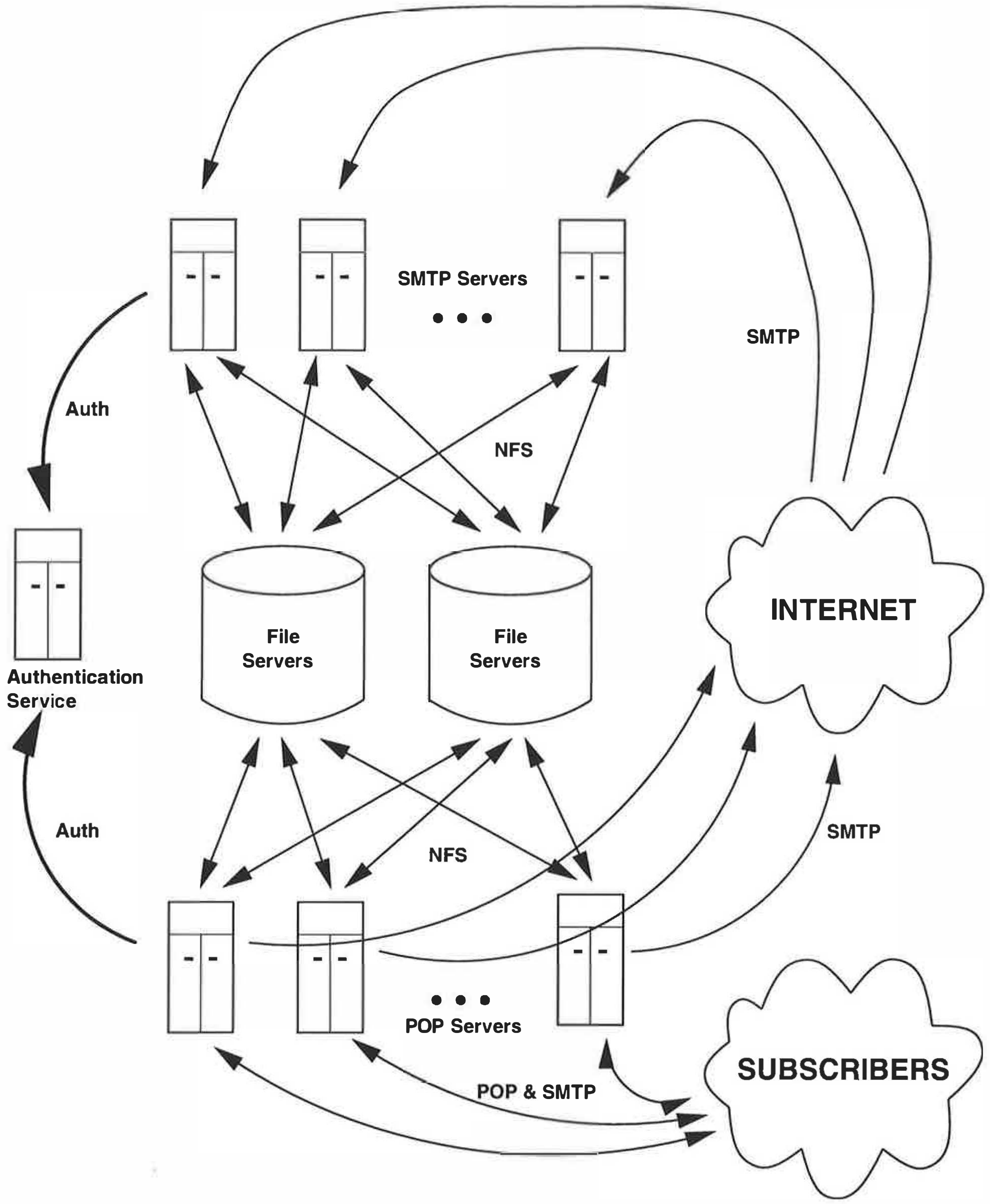

SMTP Servers
SMTP
Auth
NFS
File Servers
File Servers
INTERNET
Authentication Service
Auth
NFS
SMTP
POP Servers
SUBSCRIBERS
POP & SMTP

Figure 1

Improving Web Server Performance by Caching Dynamic Data

Arun Iyengar and Jim Challenger

IBM Research Division
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
{aruni, challngr}@watson.ibm.com

Abstract

Dynamic Web pages can seriously reduce the performance of Web servers. One technique for improving performance is to cache dynamic Web pages. We have developed the DynamicWeb cache which is particularly well-suited for dynamic pages. Our cache has improved performance significantly at several commercial Web sites. This paper analyzes the design and performance of the DynamicWeb cache. It also presents a model for analyzing overall system performance in the presence of caching. Our cache can satisfy several hundred requests per second. On systems which invoke server programs via CGI, the DynamicWeb cache results in near-optimal performance, where optimal performance is that which would be achieved by a hypothetical cache which consumed no CPU cycles. On a system we tested which invoked server programs via ICAPI which has significantly less overhead than CGI, the DynamicWeb cache resulted in near-optimal performance for many cases and 58% of optimal performance in the worst case. The DynamicWeb cache achieved a hit rate of around 80% when it was deployed to support the official Internet Web site for the 1996 Atlanta Olympic games.

1 Introduction

Web servers provide two types of data: static data from files stored at a server and dynamic data which are constructed by programs that execute at the time a request is made. The presence of dynamic data often slows down Web sites considerably. High-performance Web servers can typically deliver several hundred static files per second. By contrast, the rate at which dynamic pages are delivered is often one or two order of magnitudes slower [10].

One technique for reducing the overhead of dynamic page creation is to cache dynamic pages at the server the first time they are created. That way, subsequent requests for the same dynamic page can access the page from the cache instead of repeatedly invoking a program to generate the same page.

A considerable amount of work has been done in the area of proxy caching. Proxy caches store data at sites that are remote from the server which originally provided the data. Proxy caches reduce network traffic and latency for obtaining Web data because clients can obtain the data from a local proxy cache instead of having to request the data directly from the site providing the data. Although our cache, known as the DynamicWeb cache, can function as a proxy cache, the aspects we shall focus on in this paper are fundamentally different from those of proxy caches. The primary purpose of the DynamicWeb cache is to reduce CPU load on a server which generates dynamic pages and not to reduce network traffic. DynamicWeb is directly managed by the application generating dynamic pages. Although it is not a requirement, DynamicWeb would typically reside on the set of processors which are managing the Web site [3].

Dynamic pages present many complications which is why many proxy servers do not cache them. Dynamic pages often change a lot more frequently than static pages. Therefore, an effective method for invalidating or updating obsolete dynamic pages from caches is essential. Some dynamic pages modify state at the server each time they are invoked and should never be cached.

For many of the applications that use the DynamicWeb cache, it is essential for pages stored in the cache to be current at all times. Determining when dynamic data should be cached and when cached data has become obsolete is too difficult for the Web server to determine automatically. DynamicWeb thus provides API's for Web application programs to explicitly add and delete things from the cache. While this approach complicates the application program somewhat, the performance gains realized by applications deploying our cache have been significant. DynamicWeb has been deployed at numerous IBM and customer Web sites serving a high percentage of dynamic Web pages. We believe that its importance will continue to grow as dynamic content on the Web increases.

1.1 Previous Work

Liu [11] presents a number of techniques for improving Web server performance on dynamic pages including caching and the use of cliettes, which are long-running processes that can hold state and maintain open connections to databases that a Web server can communicate with. Caching is only briefly described. Our paper analyzes caching in considerably more detail than Liu's paper. A number of papers have been published on proxy caching [1, 4, 6, 7, 12, 13, 15, 24]. None of these papers focus on improving performance at servers generating a high percentage of dynamic pages. Gwertzman and Seltzer [8] examine methods for keeping proxy caches updated in situations where the original data are changing. A number of papers have also been published on cache replacement algorithms for World Wide Web caches [2, 18, 22, 23].

2 Cache Design

Our cache architecture is very general and allows an application to manage as many caches as it desires. The application program can choose whatever algorithm it pleases for dividing data among several caches. In addition, the same cache can be used by multiple applications.

Our cache architecture centers around a cache manager which is a long-running daemon process managing storage for one or more caches (Figure 1). Application programs communicate with the cache manager in order to add or delete items from a cache. It is possible to run multiple cache managers concurrently on the same processor by configuring each cache manager to listen for requests on a different port number. A single application can access multiple cache managers. Similarly, multiple applications can access the same cache.

The application program would typically be invoked by a Web server via the Common Gateway Interface (CGI) [21] or a faster mechanism such as

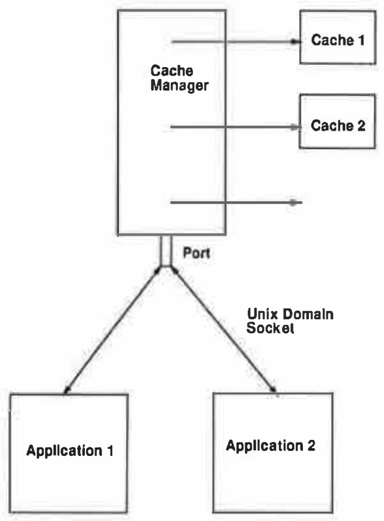


Figure 1: Applications 1 and 2 both have access to the caches managed by the cache manager. The cache manager and both applications are all on the same processor.

the Netscape Server Application Programming Interface (NSAPI) [16], the Microsoft Internet Application Programming Interface (ISAPI) [14], IBM's Internet Connection Application Programming Interface (ICAPI), or Open Market's FastCGI [17]. However, the application does not have to be Web-related. DynamicWeb can be used by other sorts of applications which need to cache data for improved performance. The current set of cache API's are compatible with any POSIX-compliant C or C++ program. Furthermore, the cache is not part of the Web server and can be used in conjunction with any Web server.

The cache manager can exist on a different node from the application accessing the cache (Figure 2). This is particularly useful in systems where multiple nodes are needed to handle the traffic at a Web site. A single cache manager running on a dedicated node can handle requests from multiple Web servers. If a single cache is shared among multiple Web servers, the costs for caching objects is reduced because the object only has to be added to a single cache. In addition, cache updates are simpler, and there are no cache coherency problems.

The cache manager can be configured to store objects in file systems, within memory buffers, or partly within memory and partly within the file system. For small caches, performance is optimized by storing objects in memory. For large caches, some objects have to be stored on disk. The cache

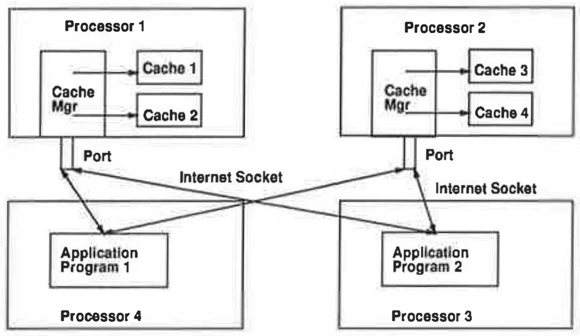


Figure 2: The cache manager and the applications accessing the caches can run on different nodes. In this situation, the cache manager and application communicate over Internet sockets.

manager is multithreaded in order to allow multiple requests to be satisfied concurrently. This feature is essential in keeping the throughput of the cache manager high when requests become blocked because of disk I/O. The cache manager achieves high throughputs via locking primitives which allow concurrent access to many of the cache manager's data structures. When the cache manager and an application reside on different nodes, they communicate via Internet sockets. When the cache manager and an application reside on the same node, they communicate via Unix Domain sockets, which are generally more efficient than Internet sockets.

The overhead for setting up a connection between an application program and a cache can be significant, particularly if the cache resides on a different node than the application program. The cache API's allow long-running connections to be used for communication between a cache manager and an application program. That way, the overhead for establishing a connection need only be incurred once for several cache transactions.

3 Cache Performance

The DynamicWeb cache has been deployed at numerous Web sites by IBM customers. While it has proved to be difficult to obtain reliable performance numbers from our customers, we have extensively measured the performance of the cache on experimental systems at the T. J. Watson Research Center. Section 3.1 presents performance measurements taken from such a system. Section 3.2 presents a method for predicting overall system performance from the performance measurements presented in Section 3.1. Section 3.3 presents cache hit rates which were observed when DynamicWeb was used at a high-volume Web site accessed by people in many different countries.

3.1 Performance Measurements from an Actual System

The system used for generating performance data in this section is shown in Figure 3. Both the cache manager and Web server were on the same node which is an IBM RS/6000 Model 590 workstation running AIX version 4.1.4.0. This machine contains a 66 Mhz POWER2 processor and comprises one node of an SP2 distributed-memory multiprocessor. The Web server was the IBM Internet Connection Secure Server (ICS) version 4.2.1. Three types of experiments were run:

1. Experiments in which requests were made to the cache manager directly from a driver program running on the same node without involving the Web server. The purpose of these experiments was to measure cache performance independently from Web server performance.

2. Experiments in which requests were made to the Web server from remote nodes running the WebStone [19] benchmark without involving the cache. The purpose of these experiments was to measure Web server performance independently of cache performance. WebStone is a widely used benchmark from Silicon Graphics, Inc. which measures the number of requests per second which a Web server can handle by simulating one or more clients and seeing how many requests per second can be satisfied during the duration of the test.

3. Experiments in which server programs which accessed the cache were invoked by requests made to the Web server from remote nodes running WebStone.

The configuration which we used is representative of a high-performance Web site but not optimal. Slightly better performance could probably be achieved by using a faster processor. There are also minor optimizations one can make to the Web server, such as turning off logging, which we didn't make. Such optimizations might have improved performance slightly. However, our goals were to use a consistent set of test conditions so that we could accurately compare the results from different experiments and to obtain good performance

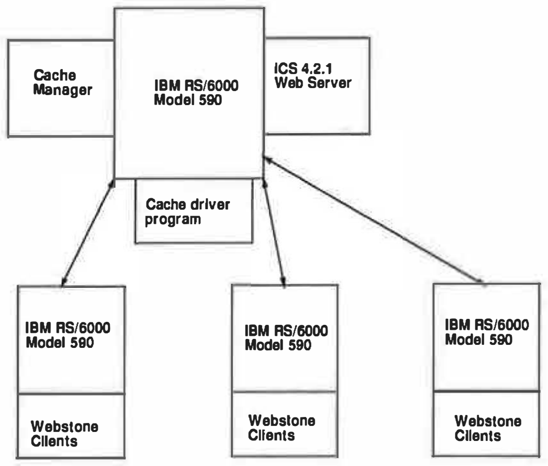


Figure 3: The system used for generating performance data.

but not necessarily the highest throughput numbers possible. Consistent test conditions are crucial, and attempts to compare different Web servers by looking at published performance on benchmarks such as WebStone and SPECweb96 [20] are often misleading because the test conditions will likely differ. Performance is affected by the hardware on which the Web server runs, software (e.g. the operating system, the TCP/IP software), and how the Web server is configured (e.g. whether or not logging is turned on).

As an example of the sensitivity of performance to different test conditions, the Web server and all of the nodes running WebStone (Figure 3) are part of an SP2. The nodes of our SP2 are connected by two networks: an Ethernet and a high-performance switch. The switch has higher bandwidth than the Ethernet. In our case, however, both the switch and the Ethernet had sufficient bandwidth to run our tests without becoming a bottleneck. One would suspect that throughput would be the same regardless of which network was used. However, we observed slightly better performance when the clients running WebStone communicated with the Web server over the switch instead of the Ethernet. This is because the software drivers for the switch are more efficient than the software drivers for the Ethernet, a fact which is unlikely to be known by most SP2 programmers. The WebStone performance numbers presented in this paper were generated using the Ethernet because the switch was frequently down on our system.

Figure 4 compares the throughput of the cache when driven by the driver program to the throughput of the Web server when driven by WebStone running on remote nodes. In both Figures 4 and 5, the cache and Web server were tested independently of each other and did not interact at all. 80% of the requests to the cache manager were read requests and the remaining 20% were write requests. The cache driver program which made requests to the cache and collected performance statistics ran on the same node as the cache and took up some CPU cycles. The cache driver program would not be needed in a real system where cache requests are made by application programs. Without the cache driver program overhead, the maximum throughput would be around 500 requests per second. The cache can sustain about 11% more read requests per second than write requests.

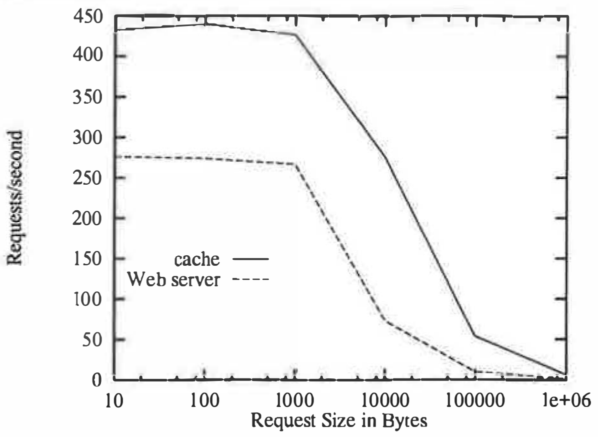


Figure 4: The throughput in requests per second which can be sustained by the cache and the Web server on a single processor. The cache driver program maintained a a single open connection for all requests. Eighty percent of requests to the cache were read requests and 20% were write requests. All requests to the Web server were for static HTML files.

In the experiments summarized in Figure 4, a single connection was opened between the cache driver program and the cache manager and maintained for the duration of the test. A naive interface between the Web server and the cache manager would make a new connection to the cache manager for each request. The first two bars of Figure 5 show the effect of establishing a new connection for each request. The cache manager can sustain close to 430 requests per second when a single open connection is maintained for all requests and about 190 requests per second when a new connection is made for each request. Since the driver program and cache manager were on the same node, Unix domain sockets were used. If they had been on different nodes, Internet sockets would have been needed, and the performance would likely have been worse.

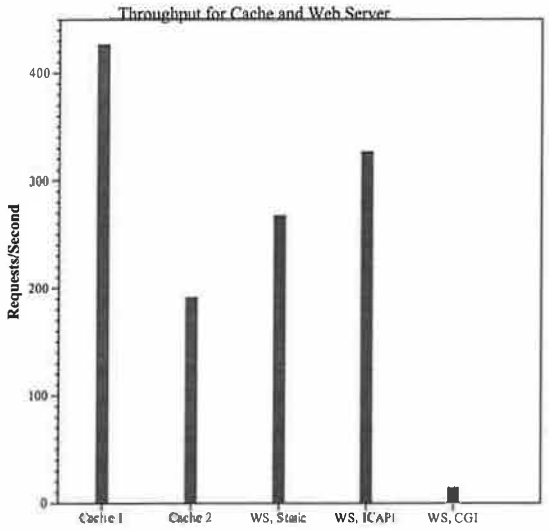


Figure 5: The throughput in requests per second which can be sustained by the cache and the Web server on a single processor under different conditions. The Cache1 bar graph represents the performance of the cache when a single long lived connection is maintained for all requests made by the driver program. The Cache2 bar graph represents the performance of the cache when a new Unix domain socket is opened for each request. The three bar graphs to the right represent the performance of the Web server.

Figure 5 also shows the performance of the Web server for different types of accesses. In both Figures 5 and 6, request sizes were less than 1000 bytes. We saw little variability in performance as a function of request size until request sizes exceeded 1000 bytes (Figure 4). For objects not exceeding 1000 bytes, the Web server can deliver around 270 static files per second. The number of dynamic pages created by very simple programs which can be returned by the ICAPI interface is higher, around 330 per second. The Common Gateway Interface (CGI) is very slow, however. Fewer than 20 dynamic pages per second can be returned by CGI, even if the programs creating the dynamic pages are very simple. The overhead of CGI is largely due to forking off a new process each time a CGI program is invoked.

ICAPI uses a programming model in which the server is multithreaded. Server programs are compiled as shared libraries which are dynamically loaded by the Web server and execute as a thread within the Web server's process. There is thus no overhead for forking off a new process when a server program is invoked through ICAPI. The ICAPI interface is fast. One of the disadvantages to ICAPI, however, is that the server program becomes part of the Web server. It is now much easier for a server program to crash the Web server than if CGI is used. Another problem is that ICAPI programs must be thread-safe. It is not always a straightforward task to convert a legacy CGI program to a thread-safe ICAPI program. Furthermore, debugging ICAPI programs can be quite challenging.

Server API's such as FastCGI use a slightly different programming model. Server programs are long-running processes which the Web server communicates with. Since the server programs are not part of the Web server's process, it is less likely for a server program to crash the Web server compared with the multithreaded approach. FastCGI programs do not have to be thread-safe. One disadvantage is that the FastCGI interface may be slightly slower than the ICAPI one because interprocess communication is required.

Using an interface such as ICAPI, it would be possible to implement our cache manager as part of the Web server which is dynamically loaded as a shared library at the time the Web server is started up. There would be no need for a separate cache manager daemon process. Cache accesses would be faster because the Web server would not have to communicate with a separate process. This kind of implementation is not possible with interfaces such as CGI or FastCGI.

We chose not to implement our cache manager in this fashion because we wanted our cache manager to be compatible with as wide a range of interfaces as possible and not just ICAPI. Another advantage of our design is that it allows the cache to be accessed remotely from many Web servers while the optimized ICAPI approach just described does not.

Figure 6 shows the performance of the Web server when server programs which access the cache are invoked via the ICAPI interface. The first bar shows the throughput when all requests to the cache manager are commented out of the server program. The purpose of this bar is to illustrate the overhead of the server program without the effect of any cache accesses. Slightly over 290 requests/second can be sustained under these circumstances. A comparison of this bar with the fourth bar in Figure 5 reveals that most of the overhead results from the ICAPI interface and not the actual work done by the server program.

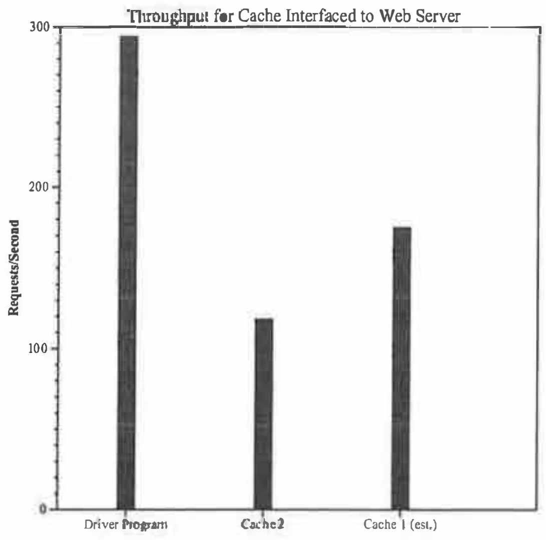


Figure 6: The throughput in requests per second for the cache interfaced to the Web server. The Driver Program bar graph is the throughput which can be sustained by the Web server running the cache driver program through the ICAPI interface with the calls to the cache manager commented out. The Cache 2 bar graph is the throughput for the same system with the cache manager calls. Each cache request from the Web server opens a new connection to the cache manager. The Cache 1 bar graph is an estimate of the throughput of the entire system if the Web server were to maintain long-lived open connections to the cache manager.

The second bar shows the performance of the Web server when each server program returns an item of 1000 bytes or less from the cache. Each request opens up a new connection with the cache manager. About 120 requests/second can be sustained. The observed performance is almost exactly what one would calculate by combining the average request time of the cache (represented by the reciprocal of the second bar in Figure 5) and the Web server driver program (represented by the reciprocal of the first bar in Figure 6) both measured independently of each other.

Performance can be improved by maintaining persistent open connections between the cache manager and Web server. That way, new connections don't have to be opened for each request. The performance one would expect to see under these circumstances is shown in the third bar of Figure 6. It is obtained by combining the average request time of the cache (represented by the reciprocal of the first bar in Figure 5) and the Web server driver program (represented by the reciprocal of the first bar in Figure 6) both measured independently of each other. The reciprocal of this quantity is the throughput of the entire system which is 175 requests/second.

3.2 An Analysis of System Performance

The throughput achieved by any system is limited by the overhead of the server program which communicates with the cache. If server programs are invoked via CGI, this overhead is generally over 20 times more than the CPU time for the cache manager to perform a single transaction. The result is that the cache manager consumes only a small fraction of the CPU time. Using a faster cache than the DynamicWeb cache would have little if any impact on overall system performance. In other words, the DynamicWeb cache results in near-optimal performance.

When faster interfaces for invoking server programs are used, the CPU time consumed by the DynamicWeb cache becomes more significant. This section presents a mathematical model of the overall performance of a system similar to the one we tested in the previous section in which server programs are invoked through ICAPI, which consumes much less CPU time than CGI. The model demonstrates that DynamicWeb achieves near-optimal system throughput in many cases. In the worst case, DynamicWeb still manages to achieve 58% of the optimal system throughput.

Consider a system containing a single processor running both a Web server and one or more cache managers. Let us assume that the performance of the system is limited by the processor's CPU. Define

h = cache hit rate expressed as the proportion of requests which can be satisfied from the cache.

s = average CPU time to generate a dynamic page by invoking a server program (i.e. CPU time for a cache miss).

c = average CPU time to satisfy a request from the cache (i.e. CPU time for a cache hit). $c = c' + c''$ where c' is the average CPU time taken up by a program invoked by the Web server for communicating with a cache manager and c'' is the average CPU time taken up by a cache manager for satisfying a request.

p_{dyn} = proportion of requests for dynamic pages

f = average CPU time to satisfy a request for a static file.

Then the average CPU time to satisfy a request on the system is:

$$T = (h * c + (1-h) * s) * p_{dyn} + f * (1 - p_{dyn}) \quad (1)$$

System performance is often expressed as *throughput* which is the number of requests which can be satisfied per unit time. Throughput is the reciprocal of the average time to satisfy a request. The throughput of the system is given by:

$$T_{tp} = \frac{1}{(\frac{h}{c_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (2)$$

where $T_{tp} = 1/T$, $c_{tp} = 1/c$, $s_{tp} = 1/s$, and $f_{tp} = 1/f$.

The number of requests per second which can be serviced from our cache manager, c_{tp}, is around 175 per second in the best case on the system we tested. Most of the overhead in such a situation results from c' because invoking server programs is costly, even using interfaces such as NSAPI and ICAPI.

The number of dynamic pages per second which can be generated by a server program, s_{tp}, varies considerably depending on the application. Values for s_{tp} as low as 1 per second are not uncommon. The overhead of the Common Gateway Interface (CGI) is enough to limit s_{tp} to a maximum of around 20 per second for any server program using this interface. In order to get higher values of s_{tp}, an interface such as NSAPI, ISAPI, ICAPI, or FastCGI must be used instead.

The rate at which static files can be served, f_{tp}, is typically several hundred per second on a high-performance system. On the system we tested, f_{tp} was around 270 per second. The proportion of requests for dynamic pages, p_{dyn}, is typically less than .5, even for Web sites where all hypertext links are dynamic. This is because many dynamic pages at such Web sites include one or more static image files.

Figures 7 shows the system throughput T_{tp} which can be achieved by a system similar to ours when all of the requests are for dynamic pages. The parameter values used by this and all other graphs in this section were obtained from the system we tested and include c_{tp} = 175 requests per second and f_{tp} = 270 requests per second. Two curves are shown for nonzero hit rates, one representing optimal T_{tp} values which would be achieved by a hypothetical system where the cache manager didn't consume any CPU cycles and another representing T_{tp} values which would be achieved by a cache similar to ours.

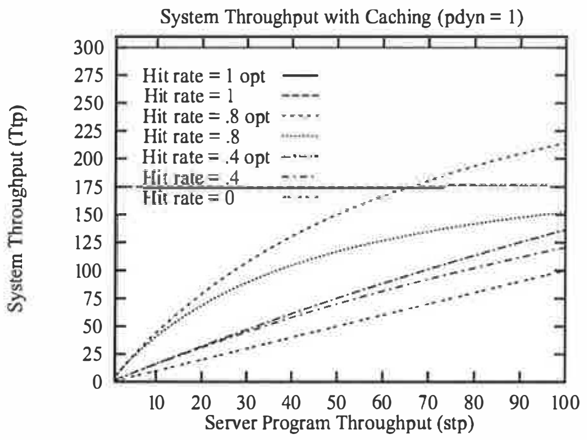


Figure 7: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when all requests are for dynamic pages. The curves with legends ending in *opt* represent hypothetical optimal systems in which the cache manager consumes no CPU cycles.

Figures 8 and 9 are analogous to Figure 7 when the proportion of dynamic pages are .5 and .2 respectively. Even Figure 9 represents a very high percentage of dynamic pages. Web sites for which almost all hypertext links are dynamic could have p_{dyn} close to .2 because of static image files embedded within dynamic pages. The official Internet Web site for the 1996 Atlanta Olympic Games (Section 3.3) is such as an example.

These graphs show that DynamicWeb often results in near optimal system throughput, particularly when the cost for generating dynamic pages is high (i.e. s_{tp} is low). This is precisely the situation when caching is essential for improved performance. In the worst case, DynamicWeb manages to achieve 58% of the optimal system performance.

Another important quantity is the *speedup*, which is the throughput of the system with caching divided by the throughput of the system without caching:

$$S = \frac{\frac{p_{dyn}}{s_{tp}} + \frac{1-p_{dyn}}{f_{tp}}}{(\frac{h}{c_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (3)$$

Figure 10 shows the speedup S which can be achieved by a system similar to ours when all of

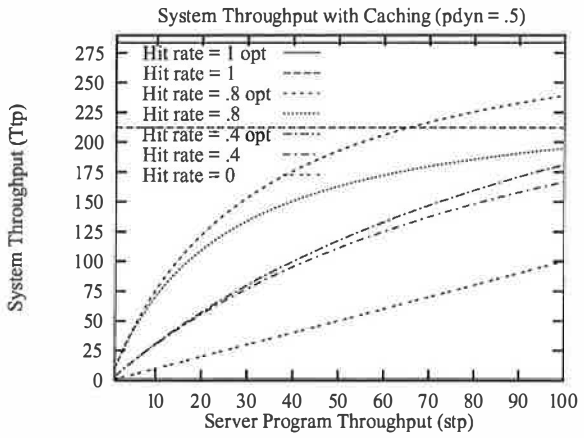


Figure 8: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when 50% of requests are for dynamic pages.

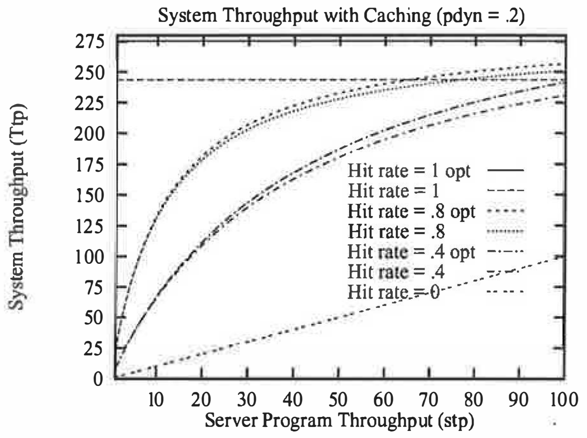


Figure 9: The throughput in connections per second (T_{cp}) achieved by a system similar to ours when 20% of requests are for dynamic pages.

the requests are for dynamic pages. Figures 11 and 12 are analogous to Figure 10 when the proportion of dynamic pages are .5 and .2 respectively. For hit rates below one, DynamicWeb achieves near optimal speedup when the cost for generating dynamic pages is high (i.e. s_{tp} is low). Furthermore, for any hit rate below 1, there is a maximum speedup which can be achieved regardless of how low s_{tp} is. This behavior is an example of Amdahl's Law [9]. The maximum speedup which can be achieved for a given hit rate is independent of the proportion of dynamic pages, p_{dyn}. However, for identical values of s_{tp}, the speedup achieved for a high value of p_{dyn} is greater than the speedup achieved for a lower value of p_{dyn} although this difference approaches 0 as s_{tp} approaches 0.

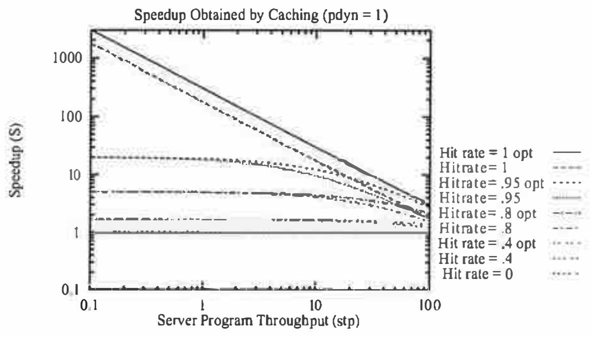


Figure 10: The speedup S achieved by a system similar to ours when all requests are for dynamic pages. The curves with legends ending in *opt* represent hypothetical optimal systems in which the cache manager consumes no CPU cycles.

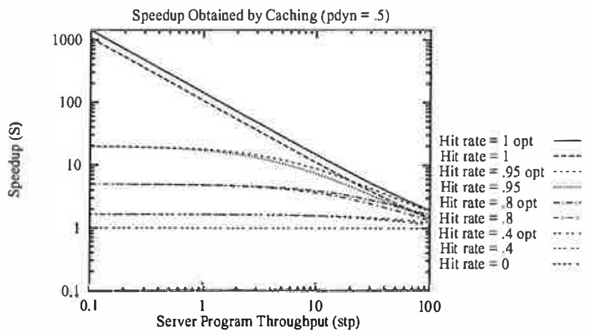


Figure 11: The speedup S achieved by a system similar to ours when 50% of requests are for dynamic pages.

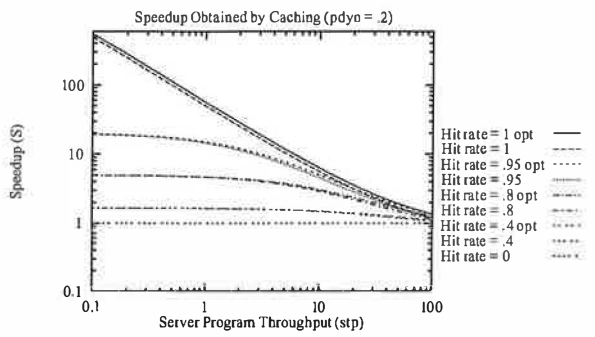


Figure 12: The speedup S achieved by a system similar to ours when 20% of requests are for dynamic pages.

3.2.1 Remote Shared Caches

In some cases, it is desirable to run the cache manager on a separate node from the Web server. An example of this situation would be a multiprocessor Web server where multiple processors each running one or more Web servers are needed to service a high-volume Web site [5]. A cache manager running on a single processor has the throughput to satisfy requests from several remote Web server nodes. One advantage to using a single cache manager in this situation is that cached data only needs to be placed in one cache. The overhead for caching new objects or updating old objects in the cache is reduced. Another advantage is that there is no need to maintain coherence among multiple caches distributed among different processors.

We need a modified version of Equation 2 to calculate the throughput of each Web server in this situation. Recall that c' is the average CPU time taken up by a program invoked by the Web server for communicating with a cache manager. Let $c'_{tp} = 1/c'$. When CGI is used, c'_{tp} is around 20 per second. Most of the overhead results from forking off a new process for each server program which is invoked. When ICAPI is used, c'_{tp} is around 300 per second, and the overhead resulting in c' is mostly due to the ICAPI interface, not the work done by the server programs. The throughput each Web server can achieve is

$$T'_{tp} = \frac{1}{(\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (4)$$

The right hand side of this equation is the same as that for Equation 2 except for the fact that c_{tp} has been replaced by c'_{tp}.

In a well-designed cache such as ours, cache misses use up few CPU cycles. The vast majority of cache manager cycles are consumed by cache hits. The throughput of cache hits for a Web server running at 100% capacity is given by

$$H_n = T'_{tp} * p_{dyn} * h = \frac{p_{dyn} * h}{(\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}}} \quad (5)$$

The number of nodes running Web servers that a single node running a DynamicWeb cache manager can service without becoming a bottleneck when all Web server nodes are running at 100% capacity is

$$N = \frac{c''_{tp}}{H_n} = \frac{c''_{tp} * ((\frac{h}{c'_{tp}} + \frac{1-h}{s_{tp}}) * p_{dyn} + \frac{1-p_{dyn}}{f_{tp}})}{p_{dyn} * h} \quad (6)$$

where $c''_{tp} = 1/c''$ (recall that c'' is the average CPU time taken by a cache manager for satisfying a request). For our system, c''_{tp} is close to 500 requests per second.

Figure 13 shows the number of nodes running Web servers that a single node running a DynamicWeb cache manager can service without becoming a bottleneck when Web server programs are invoked via CGI. It is assumed that the proportion of all Web requests for dynamic pages is .2, the Web server has performance similar to the performance of ICS 4.2.1, and the nodes in the system have performance similar to that of the IBM RS/6000 Model 590. It is also assumed that Web server nodes are completely dedicated to serving Web pages and are running at 100% capacity. If Web server nodes are performing other functions in addition to serving Web pages or are running at less than 100% capacity, the number of Web server nodes which can be supported by a single cache node increases. Figure 14 shows the analogous graph when Web server programs are invoked via ICAPI. Since ICAPI is much more efficient than CGI, Web server nodes can handle more requests per unit time and thus make more requests on the cache manager. The net result is that the cache node can support fewer Web server nodes before becoming a bottleneck.

3.3 Cache Hit Rates at a High-Volume Web Site

DynamicWeb was used to support the official Internet Web site for the 1996 Atlanta Olympic Games. This Web site received a high volume of requests from people all over the world. In order to handle the huge volume of requests which were received, several processors were utilized to provide results to the public. Each processor contained a

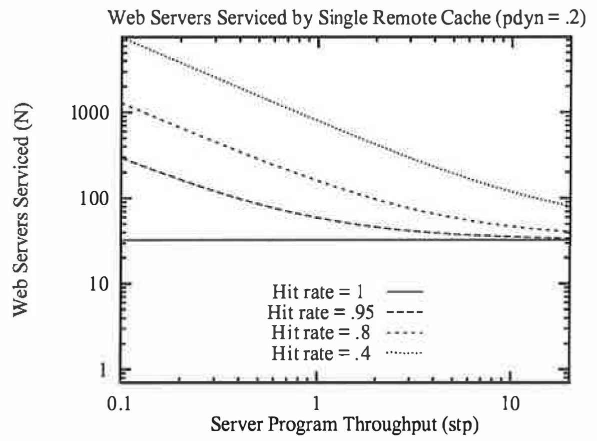


Figure 13: The number of remote Web server nodes that a single node running a DynamicWeb cache manager can service before becoming a bottleneck when Web server programs are invoked via CGI. Twenty percent of requests are for dynamic pages. Due to the overhead of CGI, s_{tp} cannot exceed 20 requests/second which is how far the X-axis extends.

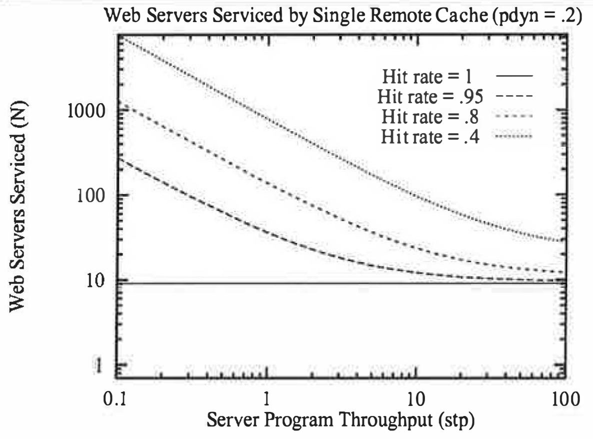


Figure 14: This graph is analogous to the one in Figure 13 when Web server programs are invoked via ICAPI.

Web server, IBM's DB2 database, and a cache manager which managed several caches. Almost all of the Web pages providing Olympics results were dynamically generated by accessing DB2. The proportion of Web server requests for dynamic pages was around .2. The remaining requests were mostly for image files embedded within the dynamic pages. Caching reduced server load considerably; the average CPU time to satisfy requests from a cache was about two orders of magnitude less than the average CPU time to satisfy requests by creating a new dynamic page.

Each cache manager managed 37 caches. Thirty-four caches were for specific sports such as badminton, baseball, and basketball. The remaining three caches were for medal standings, athletes, and schedules. Partitioning pages among multiple caches facilitated updates. When new results for a particular sport such as basketball were received by the system, each cache manager would invalidate all pages from the basketball cache without disturbing any other caches.

In order to optimize performance, cache performance monitoring was turned off for most of the Olympics. Table 1 shows the hit rates which were achieved by one of the servers when performance monitoring was enabled for a period of 2 days, 7 hours, and 40 minutes starting at 12:25 PM on July 30. The average number of read requests per second received by the cache manager during this period was just above 1.

The average cache hit rate for this period was .81. Hit rates for individual caches ranged from a high of .99 for the medal standings cache to a low of .28 for the athletes cache. Low hit rates in a cache were usually caused by frequent updates which made cached pages obsolete. Whenever the system was notified of changes which might make any pages in a cache obsolete, all pages in the cache were invalidated. A system which invalidated cached Web pages at a smaller level of granularity should have been able to achieve a better overall hit rate than .81. Since the Atlanta Olympics, we have made considerable progress in improving hit rates by minimizing the number of cached pages which need to be invalidated after a database update.

In all cases, the servers contained enough memory to store the contents of all caches with room to spare. Cache replacement policies were not an issue because there was no need to delete an object which was known to be current from a cache. Objects were only deleted if they were suspected of being obsolete.

Cache Name	*Read Requests*	*Hits*	*Hit Rate*	*Request Proportion*
Athletics	34216	25385	.74	.165
Medals	17334	17116	.99	.084
Badminton	13479	12739	.95	.065
Table Tennis	12111	11176	.92	.058
Athletes	12009	3415	.28	.058
All 37 Caches	207117	167859	.81	1.000

Table 1: Cache hit rates for the five most frequently accessed caches and all 37 caches combined. The rightmost column is the proportion of total read requests directed to a particular cache. The *Athletics* cache includes track and field sports. The *Medals* cache had the highest hit rate of all 37 caches while the *Athletes* cache had the lowest hit rate of all 37 caches.

4 Conclusion

This paper has analyzed the design and performance of the DynamicWeb cache for dynamic Web pages. DynamicWeb is better suited to dynamic Web pages than most proxy caches because it allows the application program to explicitly cache, invalidate, and update objects. The application program can ensure that the cache is up to date. DynamicWeb has significantly improved the performance of several commercial Web sites providing a high percentage of dynamic content. It is compatible with all commonly used Web servers and all commonly used interfaces for invoking server programs.

On an IBM RS/6000 Model 590 workstation with a 66 Mhz POWER2 processor, DynamicWeb could satisfy close to 500 requests/second when it had exclusive use of the CPU. On systems which invoke server programs via CGI, the DynamicWeb cache results in near-optimal performance, where optimal performance is that which would be achieved by a hypothetical cache which consumed no CPU cycles. On a system we tested in which Web servers invoked server programs via ICAPI which has significantly less overhead than CGI, the DynamicWeb cache resulted in near-optimal performance in many cases and 58% of optimal performance in the worst case. The DynamicWeb cache achieved a hit rate of around 80% when it was deployed to support the official Internet Web site for the 1996 Atlanta Olympic games.

5 Acknowledgments

Many of the ideas for the DynamicWeb cache came from Paul Dantzig. Yew-Huey Liu, Russell Miller, and Gerald Spivak also made valuable contributions.

References

[1] M. Abrams et al. Caching Proxies: Limitations and Potentials. In *Fourth International World Wide Web Conference Proceedings*, pages 119–133, December 1995.

[2] J. Bolot and P. Hoschka. Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design. *World Wide Web Journal*, pages 185–195, 1997.

[3] J. Challenger and A. Iyengar. Distributed Cache Manager and API. Technical Report RC 21004, IBM Research Division, Yorktown Heights, NY, October 1997.

[4] A. Chankhunthod et al. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–163, January 1996.

[5] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[6] C. Dodge, B. Marx, and H. Pfeiffenberger. Web cataloging through cache exploitation and steps toward Consistency Maintenance. *Computer Networks and ISDN Systems*, 27:1003–1008, 1995.

[7] S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27:165–173, 1994.

[8] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, pages 141–151, January 1996.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1996.

[10] A. Iyengar, E. MacNair, and T. Nguyen. An Analysis of Web Server Performance. In *Proceedings of GLOBECOM '97*, November 1997.

[11] Y. H. Liu, P. Dantzig, C. E. Wu, J. Challenger, and L. M. Ni. A Distributed Web Server and its Performance Analysis on Multiple Platforms. In *Proceedings of the International Conference for Distributed Computing Systems*, May 1996.

[12] A. Luotonen and K. Altis. World Wide Web proxies. *Computer Networks and ISDN Systems*, 27:147–154, 1994.

[13] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web Caching Servers Cooperate. In *Fourth International World Wide Web Conference Proceedings*, pages 107–117, December 1995.

[14] Microsoft Corporation. (ISAPI) Overview. http://www.microsoft.com/msdn/sdk/plat forms/doc/sdk/internet/src/isapimrg.htm.

[15] M. Nabeshima. The Japan Cache Project: An Experiment on Domain Cache. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[16] Netscape Communications Corporation. The Server-Application Function and Netscape Server API. http://www.netscape.com/newsref/std/server _api.html.

[17] Open Market. FastCGI. http://www.fastcgi.com/.

[18] P. Scheuermann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[19] Silicon Graphics, Inc.. World Wide Web Server Benchmarking. http://www.sgi.com/Products/WebFORCE/ WebStone/.

[20] System Performance Evaluation Cooperative (SPEC). SPECweb96 Benchmark. http://www.specbench.org/osg/web96/.

[21] Various. Information on CGI. hoohoo.ncsa.uiuc.edu:80/cgi/overview.html, www.yahoo.com/Computers/World_Wide_Web/ CGI___Common_Gateway_Interface/, www.stars.com/, and www.w3.org/pub/WWW/CGI.

[22] S. Williams et al. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of SIGCOMM '96*, pages 293–305, 1996.

[23] R. P. Wooster and M. Abrams. Proxy Caching That Estimates Page Load Delays. In *Sixth International World Wide Web Conference Proceedings*, 1997.

[24] A. Yoshida. MOWS: Distributed Web and Cache Server in Java. In *Sixth International World Wide Web Conference Proceedings*, 1997.

Measuring the Capacity of a Web Server

Gaurav Banga and Peter Druschel *

Department of Computer Science
Rice University
Houston, TX 77005

Abstract

The widespread use of the World Wide Web and related applications places interesting performance demands on network servers. The ability to measure the effect of these demands is important for tuning and optimizing the various software components that make up a Web server. To measure these effects, it is necessary to generate realistic HTTP client requests. Unfortunately, accurate generation of such traffic in a testbed of limited scope is not trivial. In particular, the commonly used approach is unable to generate client request-rates that exceed the capacity of the server being tested even for short periods of time. This paper examines pitfalls that one encounters when measuring Web server capacity using a synthetic workload. We propose and evaluate a new method for Web traffic generation that can generate bursty traffic, with peak loads that exceed the capacity of the server. Finally, we use the proposed method to measure the performance of a Web server.

1 Introduction

The explosive growth in the use of the World Wide Web has resulted in increased load on its constituent networks and servers, and stresses the protocols that the Web is based on. Improving the performance of the Web has been the subject of much recent research, addressing various aspects of the problem such as better Web caching [5, 6, 7, 23, 31], HTTP protocol enhancements [4, 20, 25, 18], better HTTP servers and proxies [2, 33, 7] and server OS implementations [16, 17, 10, 24].

To date most work on measuring Web software performance has concentrated on accurately characterizing Web server workloads in terms of request file types, transfer sizes, locality of reference in URLs requested and other related statistics [3, 5, 6, 8, 9, 12]. Some researchers have tried to evaluate the performance of Web servers and proxies using real workloads directly [13, 15]. However, this approach suffers from the experimental difficulties involved in non-intrusive measurement of a live system and the inherent irreproducibility of live workloads.

Recently, there has been some effort towards Web server evaluation through generation of synthetic HTTP client traffic, based on invariants observed in real Web traffic [26, 28, 29, 30, 1]. Unfortunately, there are pitfalls that arise in generating heavy and realistic Web traffic using a limited number of client machines. These problems can lead to significant deviation of benchmarking conditions from reality and fail to predict the performance of a given Web server.

In a Web server evaluation testbed consisting of a small number of client machines, it is difficult to simulate many independent clients. Typically, a load generating scheme is used that equates client load with the number of client processes in the test system. Adding client processes is thought to increase the total client request rate. Unfortunately, some peculiarities of the TCP protocol limit the traffic generating ability of such a naive scheme. Because of this, generating request rates that exceed the server's capacity is nontrivial, leaving the effect of request bursts on server performance unevaluated. In addition, a naive scheme generates client traffic that has little resemblance in its temporal characteristics to real-world Web traffic. Moreover, there are fundamental differences between the delay and loss characteristics of WANs and the LANs used in testbeds. Both of these factors may cause certain important aspects of Web server performance to remain unevaluated. Finally, care must be taken to ensure that limited resources in the simulated client systems do not distort the server performance results.

In this paper, we examine these issues and their effect on the process of Web server evaluation. We propose a new methodology for HTTP request generation that complements the work on Web workload modeling. Our work focuses on those aspects of the request generation method that are important for providing a scalable means of generating realistic HTTP requests, including peak

*This work was supported in part by National Science Foundation Grant CCR-9503098

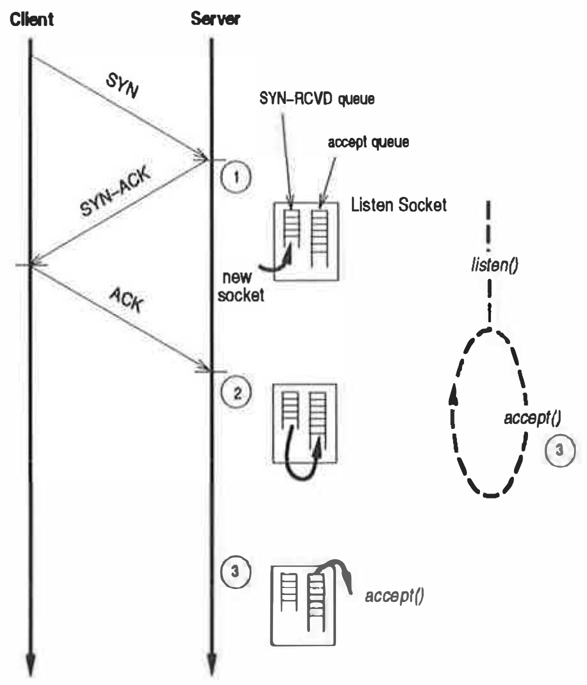


Figure 1: HTTP Connection Establishment Timeline

loads that exceed the capacity of the server. We expect that this request generation methodology, in conjunction with a representative HTTP request data set like the one used in the SPECWeb benchmark [26] and a representative temporal characterization of HTTP traffic, will result in a benchmark that can more accurately predict actual Web server performance.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the dynamics of a typical HTTP server running on a Unix based TCP/IP network subsystem. Section 3 identifies problems that arise when trying to measure the performance of such a system. In Section 4 we describe our methodology. Section 5 gives a quantitative evaluation of our methodology, and presents measurements of a Web server using the proposed method. Finally, Section 6 covers related work and Section 7 offers some conclusions.

2 Dynamics of an HTTP server

In this section, we give a brief overview of the working of a typical HTTP server on a machine with a Unix-based TCP/IP implementation. The description provides background for the discussion in the following sections. For simplicity, we focus our discussion on a BSD [14, 32] based network subsystem. The working of many other implementations of TCP/IP, such as those found in Unix System V and Windows NT, is similar.

In the HTTP protocol, for each URL fetched, a browser establishes a new TCP connection to the appropriate server, sends a request on this connection and then reads the server's response[1]. To display a typical Web page, a browser may need to initiate several HTTP transactions to fetch the various components (HTML text, images) of the page.

Figure 1 shows the sequence of events in the connection establishment phase of an HTTP transaction. When starting, a Web server process *listens* for connection requests on a socket bound to a well known port—typically port 80. When a connection establishment request (TCP SYN packet) from a client is received on this socket (Figure 1, position 1), the server TCP responds with a SYN-ACK TCP packet, creates a socket for the new, incomplete connection, and places it in the listen socket's SYN-RCVD queue. Later, when the client responds with an ACK packet to the server's SYN-ACK packet (position 2), the server TCP removes the socket created above from the SYN-RCVD queue and places it in the listen socket's queue of connections awaiting acceptance (accept queue). Each time the WWW server process executes the accept() system call (position 3), the first socket in the accept queue of the listen socket is removed and returned. After accepting a connection, the WWW server—either directly or indirectly by passing this connection to a helper process—reads the HTTP request from the client, sends back an appropriate response, and closes the connection.

In most Unix-based TCP/IP implementations, the kernel variable somaxconn limits the maximum *backlog* on a listen socket. This backlog is an upper bound on the sum of the lengths of the SYN-RCVD and accept queues. In the context of the discussion above, the server TCP drops incoming SYN packets (Figure 1, position 1) whenever this sum exceeds a value of 1.5 times the backlog[2]. When the client TCP misses the SYN-ACK packet, it goes into an exponential backoff-paced SYN retransmission mode until it either receives a SYN-ACK, or its connection establishment timer expires[3].

The average length of the SYN-RCVD queue depends on the average round-trip delay between the server and its clients, and the connection request rate. This is because a socket stays on this queue for a period of time equal to the round trip delay. Long round-trip delays and high request rates increase the length of this queue. The accept queue's average length depends on how fast the HTTP server process calls accept(), (i.e., the rate at which it serves requests,) and the request rate. If a server is operating at

[1]HTTP 1.1 supports persistent connections, but most browsers and servers today do not use HTTP 1.1.

[2]In the System V Release 4 flavors of Unix (e.g. Solaris) this sum is limited by $1 \times backlog$ rather than $1.5 \times backlog$.

[3]4.4BSD's TCP retransmits at 6 seconds and 30 seconds after the first SYN is sent before finally giving up at 75 seconds. Other TCP implementations behave similarly.

its maximum capacity, it cannot call accept() fast enough to keep up with the connection request rate and the queue grows.

Each socket's protocol state is maintained in a data structure called a Protocol Control Block (PCB). TCP maintains a table of the active PCBs in the system. A PCB is created when a socket is created, either as a result of a system call, or as a result of a new connection being established. A TCP connection is closed either actively by one of the peers executing a close() system call, or passively as a result of an incoming FIN control packet. In the latter case, the PCB is deallocated when the application subsequently performs a close() on the associated socket. In the former case, a FIN packet is sent to the peer and after the peer's FIN/ACK arrives and is ACKed, the PCB is kept around for an interval equal to the so-called TIME-WAIT period of the implementation[4]. The purpose of this TIME-WAIT state is to be able to retransmit the closing process's ACK to the peer's FIN if the original ACK gets lost, and to allow the detection of delayed, duplicate TCP segments from this connection.

A well-known problem exists with many traditional implementations of TCP/IP that limits the throughput of a Web server. Many BSD based systems have small default and maximum values for somaxconn. Since this threshold can be reached when the accept queue and/or the SYN-RCVD queue fills, a low value can limit throughput by refusing connection requests needlessly. As discussed above, the SYN-RCVD queue can grow because of long round-trip delays between server and clients, and high request rates. If the limit is too low, an incoming connection may be dropped even though the Web server may have sufficient resources to process the request. Even in the case of a long accept queue, it is usually preferable to accept a connection, unless the queue already contains enough work to keep the server busy for at least the client TCP's initial retransmission interval (6 seconds for 4.4BSD). To address this problem, some vendors have increased the maximum value of somaxconn and ship their systems with large maximum values (e.g. Digital Unix 32767, Solaris 1000). In Section 3, we will see how this fact interacts with WWW request generation.

3 Problems in Generating Synthetic HTTP requests

This section identifies problems that arise when trying to measure the performance of a Web server, using a testbed consisting of a limited number of client machines. For reasons of cost and ease of control, one would like to use a small number of client machines to simulate a large Web client population. We first describe a straightforward, commonly used scheme for generating Web traffic, and identify problems that arise.

In the simple method, a set of N Web client processes[5] execute on P client machines. Usually, the client machines and the server share a LAN. Each client process repeatedly establishes a HTTP connection, sends a HTTP request, receives the response, waits for a certain time (think time), and then repeats the cycle. The sequence of URLs requested comes from a database designed to reflect realistic URL request distributions observed on the Web. Think times are chosen such that the average URL request rate equals a specified number of requests per second. N is typically chosen to be as large as possible given P, so as to allow a high maximum request rate. To reduce cost and for ease of control of the experiment, P must be kept low. All the popular Web benchmarking efforts that we know of use a load generation scheme similar to this [26, 28, 29, 30].

Several problems arise when trying to use the simple scheme described above to generate realistic HTTP requests. We describe these problems in detail in the following subsections.

3.1 Inability to Generate Excess Load

In the World Wide Web, HTTP requests are generated by a huge number of clients, where each client has a think time distribution with large mean and variance. Furthermore, the think time of clients is not independent; factors such as human user's sleep/wake patterns, and the publication of Web content at scheduled times causes high correlation of client HTTP requests. As a result, HTTP request traffic arriving at a server is bursty with the burstiness being observable at several scales of observation [8], and with peak rates exceeding the average rate by factors of 8 to 10 [15, 27]. Furthermore, peak request rates can easily exceed the capacity of the server.

By contrast, in the simple request generation method, a small number of clients have independent think time distributions with small mean and variance. As a result, the generated traffic has little burstiness. The simple method generates a new request only after a previous request is completed. This, combined with the fact that only a limited number of clients can be supported in a small testbed, implies that the clients stay essentially in lockstep with the server. That is, the rate of generated requests never exceeds the capacity of the server.

[4]This TIME-WAIT period should be set equal to twice the Maximum Segment Lifetime (MSL) of a packet on the Internet (RFC 793[21] specifies the MSL as 2 minutes, but many implementations use a much shorter value.)

[5]In this discussion we use the terms client processes to denote either client processes or client threads, as this distinction makes no difference to our method.

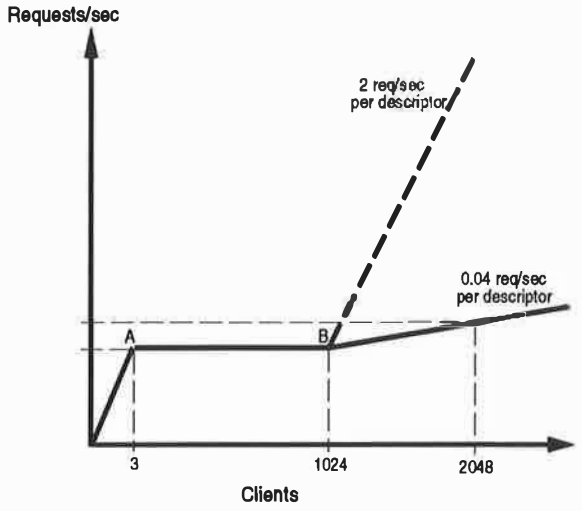


Figure 2: Request Rate versus no. of Clients

Consider a Web server that is subjected to HTTP requests from an increasing number of clients in a testbed using the simple method. For simplicity, assume that the clients use a constant think time of zero seconds, i.e., they issue a new request immediately after the previous request is completed. For small document retrievals, a small number of clients (3–5 for our test system) are sufficient to drive the server at full capacity. If additional clients are added to the system, the only effect is that the accept queue at the server will grow in size, thereby adding queuing delay between the instant when a client sees a connection as established, and the time at which the server accepts the connection and handles the request. This queuing delay reduces the rate at which an individual client issues requests. Since each client waits for a pending transaction to finish before initiating a new request, the net connection request rate of all the clients remains equal to the throughput of the server.

As we add still more clients, the server's accept queue eventually fills. At that point, the server TCP starts to drop connection establishment requests that arrive while the sum of the SYN-RCVD and accept queues is at its limit. When this happens, the clients whose connection requests are dropped go into TCP's exponential backoff and generate further requests at a very low rate. (For 4.4BSD based systems this is 3 requests in 75 seconds.) The behavior is depicted in Figure 2. The server saturates at point A, and then the request rate remains equal to the throughput of the server until the accept queue fills up (point B). Thereafter the rate increases as in the solid line at 0.04 requests/second per added client.

To generate a significant rate of requests beyond the capacity of the server, one would have to employ a huge number of client processes. Suppose that for a certain size of requested file, the capacity of a server is 100 connections/sec, and we want to generate requests at 1100 requests/sec. One would need on the order of 15000 client processes $((1100-100)/(3/75))$ beyond a number equal to the maximum size of the listen socket's accept queue to achieve this request rate. Recall from Section 2 that many vendors now configure their systems with a large value of somaxconn to avoid dropping incoming TCP connections needlessly. Thus, with somaxconn = 32767, we need 64151 processes $(1.5 \times 32767 + 15000)$to generate 1100 requests/sec. Efficiently supporting such large numbers of client processes on a small number of client machines is not feasible.

A real Web server, on the other hand, can easily be overloaded by the huge (practically infinite) client population existing on the Internet. As mentioned above, it is not at all unusual for a server to receive bursts of requests at rates that exceed the average rate by factors of 8 to 10. The effect of such bursts is to temporarily overload the server. It is important to evaluate Web server performance under overload. For instance, it is a well known fact that many Unix and non-Unix based network subsystems suffer from poor overload behavior [11, 19]. Under heavy network load these interrupt-driven systems can enter a state called *receiver-livelock*[22]. In this state, the system spends all its resources processing incoming network packets (in this case TCP SYN packets), only to discard them later because there is no CPU time left to service the receiving application programs (in this case the Web server).

Synthetic requests generated using the simple method cannot reproduce the bursty aspect of real traffic, and therefore fail to evaluate the behavior of Web servers under overload.

3.2 Additional Problems

The WAN-based Web has network characteristics that differ from the LANs on which Web servers are usually evaluated. Performance aspects of a server that are dependent on such network characteristics are not evaluated. In particular, the simple method does not model high and variable WAN delays which are known to cause long SYN-RCVD queues in the server's listening socket. Also, packet losses due to congestion are absent in LAN-based testbeds. Maltzahn et al. [13] discovered a large difference in Squid proxy performance from the idealized numbers reported in [7]. A lot of this degradation is attributed to such WAN effects, which tend to keep server resources such as memory tied up for extended periods of time.

When generating synthetic HTTP requests from a small number of client machines, care must be taken that resource constraints on the *client* machine do not accidentally distort the measured server performance. With an increasing number of simulated clients per client machine, client side CPU and memory contention are likely to arise. Eventually, a point is reached where the bottleneck in a Web transaction is no longer the server but the client. Designers of commercial Web server benchmarks have also noticed this pitfall. The WebStone benchmark [30] explicitly warns about this potential problem, but gives no systematic method to avoid it.

The primary factor in preventing client bottlenecks from affecting server performance results is to limit the number of simulated clients per client machine. In addition, it is important to use an efficient implementation of TCP/IP (in particular, an efficient PCB table[15] implementation) on the client machines, and to avoid I/O operations in the simulated clients that could affect the rate of HTTP transactions in uncontrolled ways. For example, writing logging information to disk can affect the client behavior in complex and undesirable ways. We will return to the issue of client bottlenecks in Section 4, and show how to account for client resource constraints in setting up a testbed.

4 A Scalable Method for Generating HTTP Requests

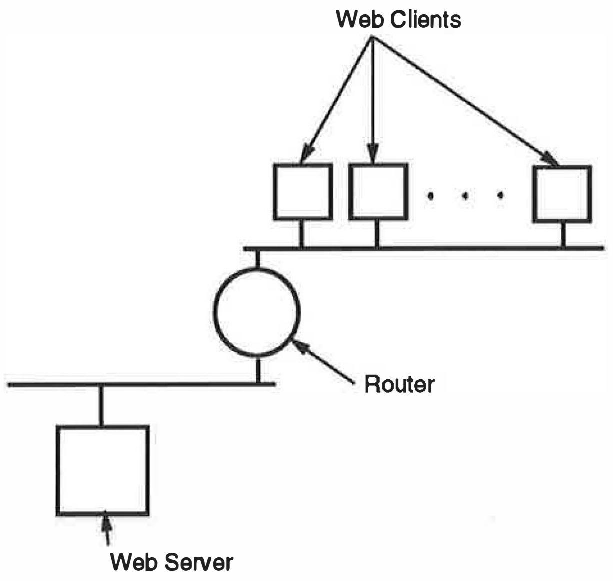


Figure 3: Testbed Architecture

In this section, we describe the design of a new method to generate Web traffic. This method addresses the problems raised in the previous section. It should be noted that our work does not by itself address the problem of accurate simulation of Web workloads in terms of the request file types, transfer sizes and locality of reference in URLs requested; instead, we concentrate on mechanisms for generating heavy concurrent traffic that has a temporal behavior similar to that of real Web traffic. Our work is intended to complement the existing work done on Web workload characterization [5, 6, 7, 23, 31], and can easily be used in conjunction with it.

4.1 Basic Architecture

The basic architecture of our testbed is shown in Figure 3. A set of P client machines are connected to the server machine being tested. Each client machine runs a number of S-Client (short for Scalable Client) processes. The structure of a S-Client, and the number of S-Clients that run on a single machine are critical to our method and are described in detail below. If WAN effects are to be evaluated, the client machines should be connected to the server through a router that has sufficient capacity to carry the maximum traffic anticipated. The purpose of the router is to simulate WAN delays by introducing an artificial delay in the router's forwarding mechanism.

4.2 S-Clients

A S-Client consists of a pair of processes connected by a Unix domain socketpair. One process in the S-Client, *the connection establishment process*, is responsible for generating HTTP requests at a certain rate and with a certain request distribution. After a connection is established, the connection establishment process sends a HTTP request to the server, then it passes on the connection to the *connection handling process*, which handles the HTTP response.

The connection establishment process of a S-Client works as follows: The process opens D connections to the server using D sockets in non-blocking mode. These D connection requests are spaced out over T milliseconds. T is required to be larger than the maximal round-trip delay between client and server (remember that an artificial delay may be added at the router).

After the process executes a non-blocking connect() to initiate a connection, it records the current time in a variable associated with the used socket. In a tight loop, the process checks if for any of its D active sockets, the connection is complete, or if T milliseconds have elapsed since a connect() was performed on this socket. In the former case, the process sends a HTTP request on the newly established connection, hands off this connection to the other process of the S-Client through the Unix domain socketpair, closes the socket, and then initiates another connection to the server. In the latter case, the process simply closes the socket and initiates another connection to the server. Notice that closing the socket in

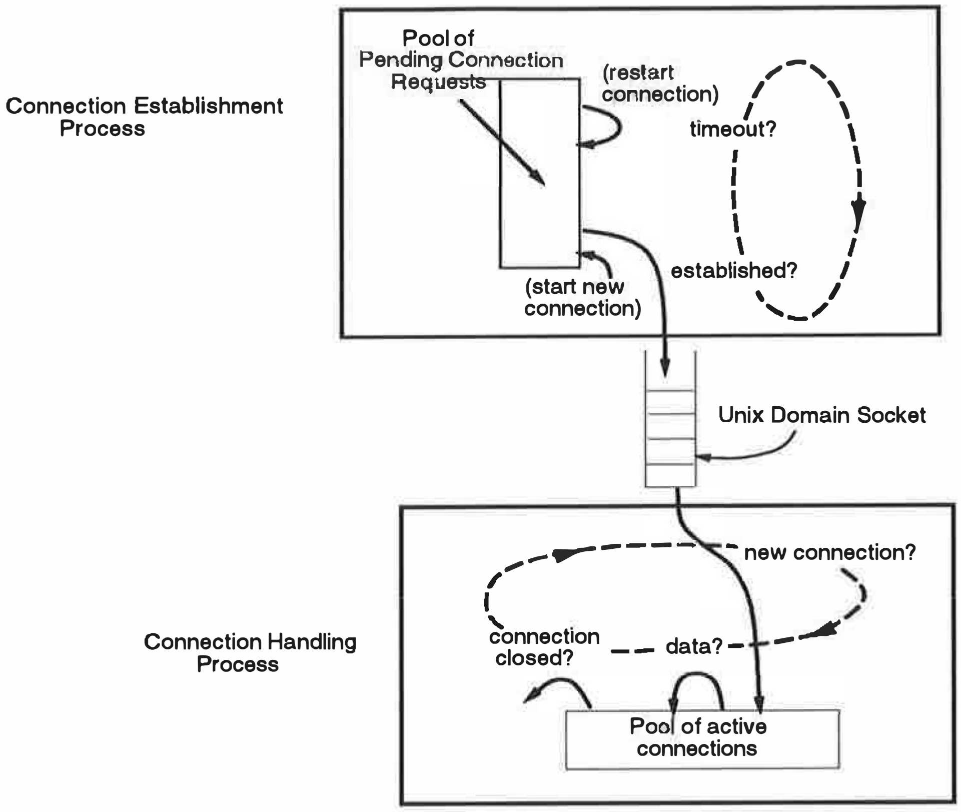

Connection Establishment Process
Pool of Pending Connection Requests
(restart connection)
timeout?
established?
(start new connection)
Unix Domain Socket
Connection Handling Process
new connection?
connection closed?
data?
Pool of active connections

Figure 4: A Scalable Client

both cases does not generate any TCP packets on the network. In effect, it prematurely aborts TCP's connection establishment timeout period. The close merely releases socket resources in the OS.

The connection handling process of a S-Client waits for 1) data to arrive on any of the active connections, or 2) for a new connection to arrive on the Unix domain socket connecting it to the other process. In case of new data on an active socket, it reads this data; if this completes the server's response, it closes the socket. A new connection arriving at the Unix domain socket is simply added to the set of active connections.

The rationale behind the structure of a S-Client is as follows. The two key ideas are to (1) shorten TCP's connection establishment timeout, and (2) to maintain a constant number of unconnected sockets (simulated clients) that are trying to establish new connections. Condition (1) is accomplished by using non-blocking connects and closing the socket if no connection was established after T seconds. The fact that the connection establishment process tries to establish another connection immediately after a connection was established ensures condition (2).

The purpose of (1) is to allow the generation of request rates beyond the capacity of the server with a reasonable number of client sockets. Its effect is that each client socket generates SYN packets at a rate of at least $1/T$. Shortening the connection establishment to $500ms$ by itself would cause the system's request rate to follow the dashed line in Figure 2.

The idea behind (2) is to ensure that the generated request rate is independent of the rate at which the server handles requests. In particular, once the request rate matches the capacity of the server, the additional queuing delays in the server's accept queue no longer reduce the request rate of the simulated clients. Once the server's capacity is reached, adding more sockets (descriptors) increases the request rate at $1/T$ requests per descriptor, eliminating the flat portion of the graph in Figure 2.

To increase the maximal request generation rate, we can either decrease T or increase D. As mentioned before, T must be larger than the maximal round-trip time between client and server. This is to avoid the case where the client aborts an incomplete connection in the SYN-RCVD state at the server, but whose SYN-ACK from the server (see Figure 1) has not yet reached the client. Given a value of T, the maximum value of D is usually limited by OS-imposed restrictions on the maximum number of open descriptors in a single process. However, depending on the capacity of the client machine, it is possible that one S-Client with a large D may saturate the client machine.

Therefore, as long as the client machine is not saturated, D can be as large as the OS allows. When multiple S-Clients are needed to generate a given rate, the largest allowable value of D should be used, as this keeps the total number of processes low, thus reducing overhead due to context switches and memory contention between the various S-Client processes. How to determine the maximum rate that a single client machine can safely generate without risking distortion of results due to client side bottlenecks is the subject of the next section.

4.3 Request Generating Capacity of a Client Machine

As noted in the previous section, while evaluating a Web server, it is very important to operate client machines in load regions where they are not limiting the observed performance. Our method for finding the maximum number of S-Clients that can be safely run on a single machine—and thus determine the value of P needed to generate a certain request rate—is as follows. The work that a client machine has to do is largely determined by the sum of the number of sockets D of all the S-Clients running on that machine. Since we do not want to operate a client near its capacity, we choose this value as the largest number N for which the throughput vs. request rate curve when using a single client machine is unchanged from the same curve when using 2 client machines. The corresponding number of S-Clients we need to use is found by distributing these N descriptors into as few processes as the OS permits. We call the request rate generated by these N descriptors the maximum raw request rate of a client machine.

It is possible that a single process's descriptor limit (imposed by the OS) is smaller than the average number of simultaneous active connections in the connection handling process of a S-Client. In this case we have no option but to use a larger number of S-Clients with smaller D values to generate the same rate. Due to increased memory contention and context switching, this may actually cause a lower maximum raw request rate for a client machine than if the OS limit on the number of descriptors per process was higher. Because of this, the number of machines needed to generate a certain request rate may be higher in this case.

4.4 Think Time Distributions

The presented scheme generates HTTP requests with a trivial think time distribution, i.e., it uses a constant think time chosen to achieve a certain constant request rate. It is possible to generate more complex request processes by adding appropriate think periods between the point where a S-Client detects a connection was established and when it next attempts to initiate another connection. In this way, any request arrival process can be generated whose peak request rate is lower than or equal to the maximum raw request rate of the system. In particular, the system can be parameterized to generate self-similar traffic [8].

5 Quantitative Evaluation

In this section we present experimental data to quantify the problems identified in Section 3, and to evaluate the performance of our proposed method. We measure the request generation limitations of the naive approach and evaluate the S-Client based request generation method proposed in Section 4. We also measure the performance of a Web server using our method.

5.1 Experimental Setup

All experiments were performed in a testbed consisting of 4 Sun Microsystems SPARCstation 20 model 61 workstations (60MHz SuperSPARC+, 36KB L1, 1MB L2, SPECint92 98.2) as the client machines. The workstations are equipped with 32MB of memory and run SunOS 4.1.3_U1. Our server is a dual processor SPARCStation 20 constructed from 2 erstwhile SPARCStation 20 model 61 machines. This machine has 64MB of memory and runs Solaris 2.5.1. A 155 Mbit/s ATM local area network connects the machines, using FORE Systems SBA-200 network adaptors. For our HTTP server, we used the NCSA httpd server software, revision 1.5.1. In our experiments we used no artificial delay in the router connecting the clients and the server. We have not yet quantitatively evaluated the effect of WAN delays on server performance.

The server's OS kernel was tuned using Web server performance enhancing tips advised by Sun. That is, we increased the total pending connections (accept+SYN-RCVD queues) limit to 1024 and decreased the TIME-WAIT period to 3 seconds.

5.2 Request generation rate

The purpose of our first experiment is to quantitatively characterize the limitations of the simple request generation scheme described in Section 3. We ran an increasing number of client processes distributed across 4 client machines. Each client tries to establish a HTTP connection to the server, sends a request, receives the response and then repeats the cycle. Each HTTP request is for a single file of size 1294 bytes. We measured the request rate (incoming SYNs/second) at the server.

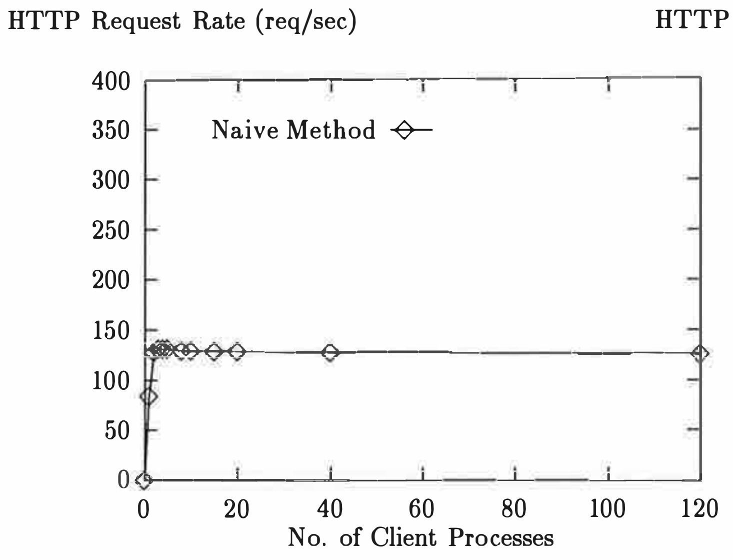


Figure 5: Request rate versus number of clients

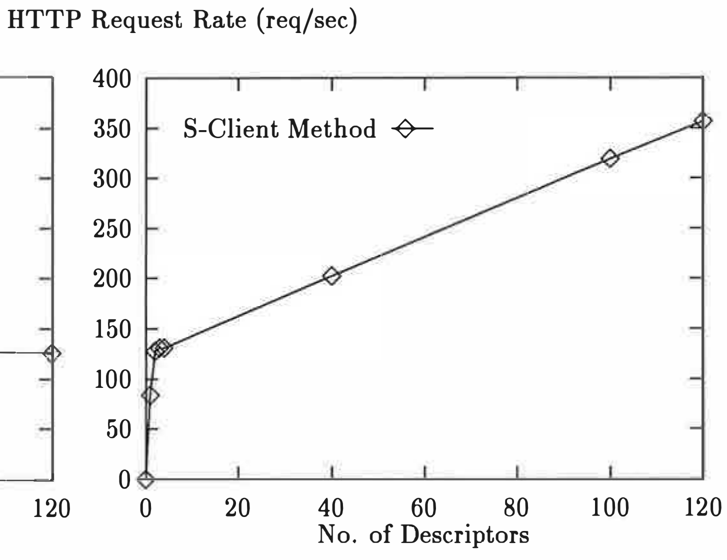


Figure 6: Request rate versus number of descriptors

In a similar test we ran 12 S-Clients distributed across the 4 client machines with an increasing number of descriptors per S-Client and measured the request rate seen at the server. Each S-Client had the connection establishment timeout period T set to $500ms$. The same file was requested as in the case of the simple clients.

Figure 5 plots the total connection request rate seen by the server versus the total number of client processes for the simple client test. Figure 6 plots the same metric for the S-Client test, but with the total number of descriptors in the S-Clients on the x-axis.

For the reasons discussed earlier, the simple scheme generates no more than about 130 requests per second (which is the capacity of our server for this request size). At this point, the server can accept connections at exactly the rate at which they are generated. As we add more clients, the queue length at the accept queue of the server's listen socket increases and the request rate remains nearly constant at the capacity of the server.

With S-Clients, the request rate increases linearly with the total number of descriptors being used for establishing connections by the client processes. To highlight the difference in behavior of the two schemes in this figure, we do not show the full curve for S-Clients. The complete curve shows a linear increase in request rate all the way up to 2065 requests per second with our setup of four client machines. Beyond this point, client capacity resource limitations set in and the request rate ceases to increase. More client machines are needed to achieve higher rates. Thus we see that S-Clients enable the generation of request loads that greatly exceed the capacity of the server. The generated load also scales very well with the number of descriptors being used.

5.3 Overload Behavior of a Web Server

Being able to reliably generate high request rates, we used the new method to evaluate how a typical commercial Web server behaves under high load. We measured the HTTP throughput achieved by the server in terms of transactions per second. The same 1294 byte file as before was used in this test.

Figure 7 plots the server throughput versus the total connection request rate. As before, the server saturates at about 130 transactions per second. As we increase the request rate beyond the capacity of the server, the server throughput declines, initially somewhat slowly, and then more rapidly reaching about 75 transactions/second at 2065 requests/second. This fall in throughput with increasing request rate is due to the CPU resources spent on protocol processing for incoming requests (SYN packets) that are eventually dropped due to the backlog on the listen socket (the full accept queue).

The slope of the throughput drop corresponds to about 325 usec worth of processing time per SYN packet. While this may seem large, it is consistent with our observation of the performance of a server system based on a 4.4BSD network subsystem retrofitted into SunOS 4.1.3_U1 on the same hardware.

This large drop in throughput of an overloaded server highlights the importance of evaluating the overload behavior of a Web server. Note that it is impossible to

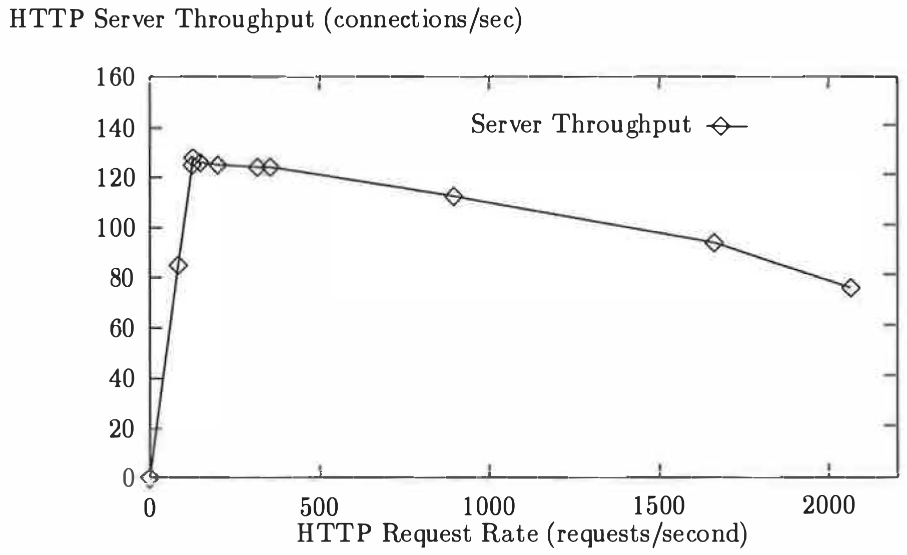


Figure 7: Web server throughput versus request rate

evaluate this aspect of Web server performance with current bench marks that are based on the simple scheme for request generation.

5.4 Throughput under Bursty Conditions

In Section 3, we point out that one of the drawbacks of the naive traffic generation scheme is the lack of burstiness in the request traffic. A burst in request rate may temporarily overload the server beyond its capacity. Since Figure 7 indicates degraded performance under overload, we were motivated to investigate the performance of a Web server under bursty conditions.

We configured a S-Client with think times values such that it generates bursty request traffic. We characterize the bursty traffic by 2 parameters, a) the ratio between the maximum request rate and the average request rate, and b) the fraction of time for which the request rate exceeded the average rate. Whenever the request rate is above the mean, it is equal to the maximum. The period is 100 seconds. For four different combination of these parameters we varied the average request rate and measured the throughput of the server. Figure 8 plots the throughput of the Web server versus the average request rate. The first parameter in the label of each curve is the factor a) above, and the second is factor b) above, expressed as a percentage. For example, (6, 5) refers to the case where for 5% of the time the request rate is 6 times the average request rate.

As expected, even a small amount of burstiness can degrade the throughput of a Web server. For the case with 5% burst ratio and peak rate 6 times the average, the throughput for average request rates well below the server's capacity is degraded by 12-20%. In general, high burstiness both in parameter a) and in b) degrades the throughput substantially. This is to be expected given the reduced performance of a server beyond the saturation point in Figure 7.

Note that our workload only approximates what one would see on the real WWW. The point of this experiment is to show that the use of S-Clients *enables* the generation of request distributions of complex nature and with high peak rates. This is not possible using a simple scheme for request generation. Moreover, we have shown that the effect of such burstiness on server performance is significant.

6 Related Work

There is much existing work towards characterizing the invariants in WWW traffic. Most recently, Arlitt and Williamson [3] characterized several aspects of Web server workloads such as request file type distribution, transfer sizes, locality of reference in the requested URLs and related statistics. Crovella and Bestavros [8] looked at Self-Similarity in WWW traffic. The invariants reported by these efforts have been used in evaluating the performance of Web servers, and the many methods proposed by researchers to improve WWW performance.

Web server benchmarking efforts have much more recent origins. SGI's WebStone [30] was one of the earliest Web server benchmarks and is the de facto industry standard, although there have been several other efforts [28, 29]. WebStone is very similar to the simple scheme

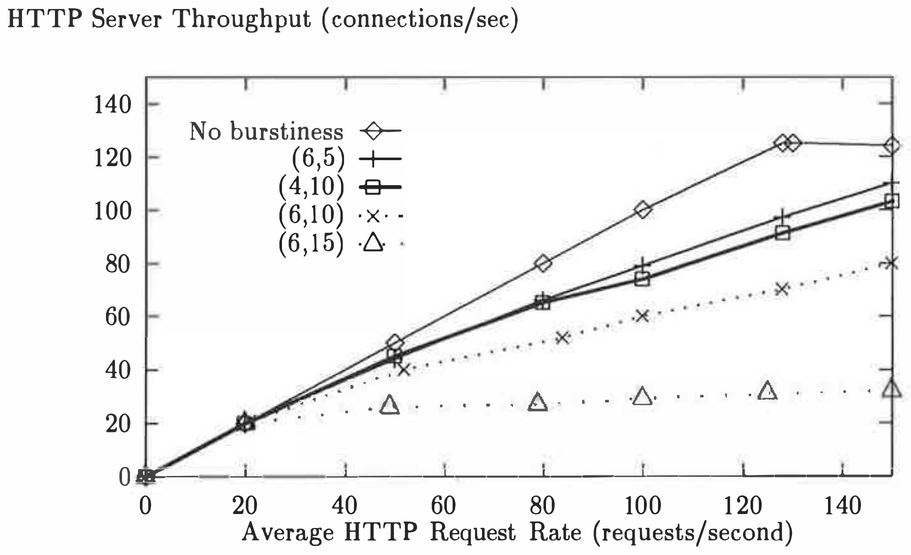


Figure 8: Web server throughput under bursty conditions versus request rate

that we described in Section 3 and suffers from its limitations. Recently SPEC has released SPECWeb96 [26], which is a standardized Web server benchmark with a workload derived from the study of some typical servers on the Internet. The request generation method of this benchmark is also similar to that of the simple scheme and so it too suffers from the same limitations.

In summary, all Web benchmarks that we know of evaluate Web Servers only by modeling aspects of server workloads that pertain to request file types, transfer sizes and locality of reference in URLs requested. No benchmark we know of attempts to accurately model the effects of request overloads on server performance. Our method based on S-Clients enables the generation of HTTP requests with burstiness and high rates. It is intended to complement the workload characterization efforts to evaluate Web servers.

7 Conclusion

This paper examines pitfalls that arise in the process of generating synthetic Web server workloads in a testbed consisting of a small number of client machines. It exposes the limitations of the simple request generation scheme that underlies state-of-the-art Web server benchmarks. We propose and evaluate a new strategy that addresses these problems using a set of specially constructed client processes. Initial experience in using this method to evaluate a typical Web server indicates that measuring Web server performance under overload and bursty traffic conditions gives new and important insights in Web server performance. Our new methodology enables the generation of realistic, bursty HTTP traffic and thus the evaluation of an important performance aspect of Web servers.

Source code and additional technical information about S-Clients can be found at http://www.cs.rice.edu/CS/Systems/Web-measurement/.

References

[1] J. Almeida, V. Almeida, and D. Yates. Measuring the Behavior of a World-Wide Web Server. Technical Report TR-96-025, Boston University, CS Dept., Boston MA, 1996.

[2] Apache. http://www.apache.org/.

[3] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[5] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad. Application-Level Document Caching in the Internet. Technical Report TR-95-002, Boston University, CS Dept., Boston MA, Feb. 1995.

[6] H. Braun and K. Claffy. Web Traffic Characterization: An Assessment of the Impact of Caching

Documents from NCSA's Web Server. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[8] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[9] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-Based Traces. Technical Report TR-95-010, Boston University, CS Dept., Boston MA, 1995.

[10] DIGITAL UNIX Tuning Parameters for Web Servers. http://www.digital.com/info/internet/document/ias/tuning.html.

[11] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[12] T. T. Kwan, R. E. McGrath, and D. A. Reed. User access patterns to NCSA's World-wide Web server. Technical Report UIUCDCS-R-95-1934, Dept. of Computer Science, Univ. IL., Feb. 1995.

[13] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[14] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[15] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[16] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[17] J. C. Mogul. Personal communication, Oct. 1996.

[18] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the SIGCOMM '97 Conference*, Cannes, France, Sept. 1997.

[19] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.

[20] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[21] J. B. Postel. Transmission Control Protocol. RFC 793, Sept. 1981.

[22] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proc. Globecom'92 IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, Dec. 1992.

[23] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[24] Solaris 2 TCP/IP. http://www.sun.com/sunsoft/solaris/networking/tcpip.html.

[25] M. Spasojevic, M. Bowman, and A. Spector. Using a Wide-Area File System Within the World-Wide Web. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[26] SPECWeb96. http://www.specbench.org/osg/web96/.

[27] W. Stevens. *TCP/IP Illustrated Volume 3*. Addison-Wesley, Reading, MA, 1996.

[28] Web 66. http://web66.coled.umn.edu/gstone/info.html.

[29] WebCompare. http://webcompare.iworld.com/.

[30] WebStone. http://www.sgi.com/Products/WebFORCE/Resources/res_webstone.html.

[31] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the SIGCOMM '96 Conference*, Palo Alto, CA, Aug. 1996.

[32] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[33] Zeus. http://www.zeus.co.uk/.

BIT: A Tool for Instrumenting Java Bytecodes

Han Bok Lee
hanlee@cs.colorado.edu
Department of Computer Science
University of Colorado, Boulder 80309
Benjamin G. Zorn
zorn@cs.colorado.edu
Department of Computer Science
University of Colorado, Boulder 80309

Abstract

BIT (Bytecode Instrumenting Tool) is a collection of Java classes that allow one to build customized tools to instrument Java Virtual Machine (JVM) bytecodes. Because understanding program behavior is an essential part of developing effective optimization algorithms, researchers and software developers have built numerous tools that carry out program analysis. Although there are existing tools that analyze and modify executables on a variety of operating systems and machine architectures, there currently is no framework for carrying out the same task for JVM bytecodes. In this paper, we describe BIT, which allows the user to insert calls to analysis methods anywhere in the bytecode, so that information can be extracted from the user program while it is being executed. In this paper, we describe several simple tools built using BIT and also report on BIT's performance. We found that the overhead for the execution speed and size were between 23% to 150%.

1. Introduction

It is often important for software developers and researchers to be able to measure and understand both the static structure and dynamic behavior of a program. Such information is used to identify critical pieces of code; for debugging purposes; to evaluate and compare the performance of different software or hardware implementations such as branch prediction, cache replacement, and instruction scheduling; and in support of profile-driven optimizations [31]. Over the years, researchers have built numerous tools that allow them to obtain this information.

This paper describes BIT (Bytecode Instrumenting Tool) [20], a tool that allows JVM bytecodes to be instrumented for the purpose of extracting measurements of their dynamic behavior. The JVM is an abstract machine specification designed to support the Java programming language, and JVM bytecodes are equivalent to binaries on other machines [21]. Although there are tools that allow binary instrumentation on a number of different operating systems and machine architectures, BIT is the first framework of which we are aware that supports JVM bytecode instrumentation. BIT is a set of Java classes that allow the user to observe the dynamic behavior of programs by inserting calls to user analysis methods at any point in the bytecode execution. Because BIT is written in Java, tools written using it are portable across platforms. Also, there are other programming languages that can be compiled into JVM bytecodes such as Kawa [6], which compiles Scheme code into JVM bytecodes and AppletMagic [17], which translates Ada 95 to JVM bytecodes. Furthermore, Hardwick and Sipelstein studied the feasibility of using Java as an intermediate language [14]. Because BIT instruments JVM bytecodes, it can be used to instrument programs written in any language that has been compiled to the JVM, and instrumentation does not require that the program source code be available.

In this paper we describe the design and implementation of BIT, present example tools built using BIT, and describe results based on measuring BIT's overhead on five Java programs, including BIT itself. BIT's instrumentation makes the instrumented JVM executables both larger and longer running. We found that the overhead for the execution speed and code size ranged from 23% to 150%.

This paper has the following organization. In Section 2, we discuss related work. In Section 3, we describe BIT's design and introduce a sample tool written using BIT. In Section 4, we discuss some details of the implementation and in Section 5 we present performance results based on instrumenting several Java applications. Finally, in Section 6, we summarize the paper and discuss future directions for research.

2. Related Work

There are many tools that employ techniques based on program instrumentation to carry out different tasks. These tasks range from emulation and tracing to optimization [19]. The Wisconsin Wind Tunnel architecture simulator [27], for example, allows the emulation of a cycle counter, which the underlying hardware does not provide. Techniques based on program instrumentation have also been used in optimizations [31, 32]. However, most of the tools that have been developed using program instrumentation techniques are used for studying program or system behavior. Tools such as QPT [18], Pixie [28], and Epoxie [33] generate address traces and instruction counts by rewriting program executables. MPTRACE [13] and ATUM [1] generate data and instruction traces, and PROTEUS [5] and Shade [7] emulate other architectures. Also, software testing and quality assurance tools that detect memory leaks and access errors such as Purify [15] catch programming errors by using these techniques. Purify inserts instructions directly into the object code produced by existing compilers. These instructions, in turn, check the validity of every memory access performed by the program and report when there are errors.

There are limitations to these tools, however, since they are designed for a specific task and are difficult if not impossible to modify to meet users' changing needs. It would be difficult, for example, for a user to modify a customized tool to obtain more or less detailed information about a trace than what is already provided. To modify a customized tool, a user has to have access to the source code and have a good understanding of how the tool works, including low-level details that deal directly with modifying the binaries. Moreover, many of these tools use inter-process communication or files to relay program behavior to the analysis routines, which are expensive [30].

There is another group of tools, sometimes called binary editing or executable editing tools, which have different design goals and offer a library of routines for modifying executable files. The tools in this group include the OM system [32], EEL [19], ATOM [30], and Etch [4], which are explained in more detail below. These tools differ in that they offer a library of routines for modifying executable files. Users, in turn, can design and build their own customized tools to meet their needs using these tools.

OM works on object files, and it represents machine instructions as Register Transfer language (RTL), which can later be manipulated and written back to the disk in the form of machine instructions. EEL also uses an intermediate representation to represent machine instructions. The difference between OM and EEL is that while OM uses relocation information in the object files to relocate edited code, EEL analyzes and modifies the program's instructions directly. Furthermore, EEL can edit fully linked executables and emphasizes portability in its design. However, EEL currently works only on workstations with SPARC processors, under Solaris and SunOS, and therefore its claim of portability is yet to be realized.

ATOM provides a framework on top of OM, and a number of customized tools from basic block counting to cache simulators can be built on top of that framework. ATOM, unlike OM and EEL, does not allow one to arbitrarily modify the object code, but simplifies the instrumentation process by providing an API to access program constructs such as procedures, basic blocks, and instructions, and it also provides a library to easily manipulate those constructs. These library routines include operations such as iterating through these constructs and inserting procedure calls before and after them. However, ATOM does not allow removing or replacing existing instructions in the binary files as EEL does. Another drawback of ATOM is that it is not portable. Currently, ATOM runs only on the Alpha AXP under OSF/1.

Etch is an application program performance evaluation and optimization system running on Intel x86 platforms running the Windows/NT operating system. Etch allows the user to instrument existing binaries with arbitrary instructions.

The tools mentioned above operate on object codes for a variety of operating systems and architectures, but none of them work on JVM class files. However, the Java interpreter provides some profiling information, which includes the method invocation sequence and the size of objects allocated, when invoked with th*prof* switch. There is a tool called NetProf [26] that visualizes Java profile information by translating Java bytecodes to Java source code. There are also tools that carry out post-processing on class files such as osjcfp [25] and the work by Cattell [23] to make classes persistence-capable. However, the inner workings of these tools have not been published. Furthermore, these are also customized tools and have the same limitations mentioned above.

BIT follows ATOM's design by providing a set of classes that users can employ to build their own program analysis tools for JVM bytecodes.

3. BIT Architecture

This section describes the design of BIT at a high level and illustrates how BIT is used with an example. Like ATOM, the architecture of BIT is based on the observation that many of the dynamic behaviors of a program can be obtained by instrumenting a few key locations, e.g., before and after methods, before and after basic blocks, and before and after instructions. Thus, BIT provides classes and methods for inserting a method invocation at each of these key locations.

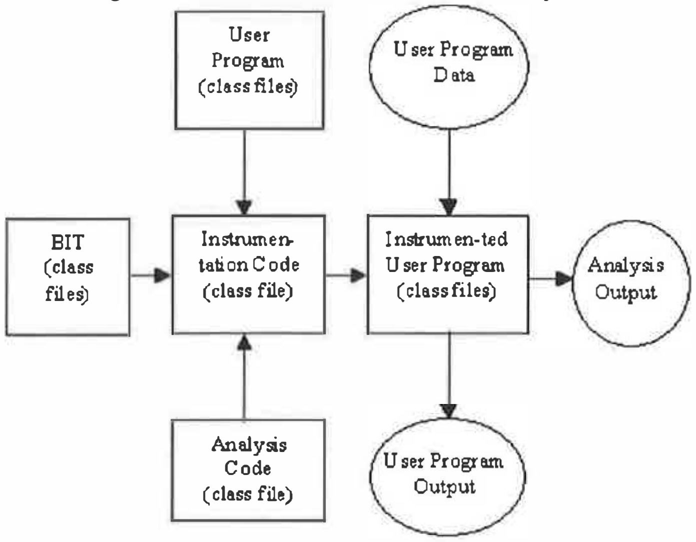

User Program (class files)
User Program Data
BIT (class files)
Instrumentation Code (class file)
Instrumen-ted User Program (class files)
Analysis Output
Analysis Code (class file)
User Program Output

Figure 1. The Process of Using BIT – Instrumentation Code uses BIT classes to read in User Program and to insert calls to Analysis Code to produce Instrumented User Program.

Figure 1 illustrates the process of using BIT. BIT is a set of Java classes that are represented by the BIT box. The user's application is compiled into JVM bytecodes with a Java compiler. The User Program box represents the application output. The user writes personalized instrumentation code by using the classes and methods that BIT provides. The user also writes analysis code. Both instrumentation and analysis code are compiled into JVM bytecodes by a Java compiler and are represented by Instrumentation and Analysis Code boxes respectively. When the JVM executes the instrumentation code, it will read in the user program, which is in the form of JVM bytecodes, and insert calls to the analysis code at appropriate places in the user program. This process results in the instrumented user program, which then can be executed under the JVM to produce both the original program output, which is represented by the User Program Output circle, and the analysis output, which is represented by the Analysis Output circle. The inserted calls to the analysis routines do not have any semantic effect on the instrumented program, which should produce exactly the same output as the original program.

To provide a concrete understanding of how BIT works, we present a customized tool that could aid in branch prediction as illustrated by Srivastava and Eustace [30]. This tool counts the number of branches taken and not taken at all the branches in different methods. Figure 2 shows the instrumentation code for this tool. BIT's methods are shown in bold. This instrumentation code specifies where the user program is to be instrumented and what methods are to be invoked. This tool takes two arguments: an input file and an output file. The input class file is opened and broken down into more manageable pieces (class, routine, basic block, and instruction), which are then instrumented. In this instrumentation program, we first analyze the input file by creating a new *ClassInfo* object, whose constructor parses the input file and stores the intermediate representation in its members. A call to the **getRoutines**() method is invoked to obtain the vector of Routines. A *Routine* represents a method in the input class file. For each of these *Routines*, we obtain the basic blocks by invoking the **getBasicBlocks**() method. To count all the conditional branches in a program, we need to insert analysis code wherever there exists a conditional instruction in the program. This is accomplished by looking at each basic block and seeing whether the last instruction in that basic block is a conditional instruction or not.

Once we find conditional instructions, we insert calls to analysis methods before these conditional instructions are executed. In addition, calls are made when entering and leaving a method so that variables are initialized and results are printed.

Figure 3 shows the analysis code that is invoked when conditional instructions are encountered. The **LeaveMethod**() method prints the statistics gathered during this method's execution. The **Offset**() method puts the offset of the conditional instruction being executed into a static variable named *pc*, and the **Branch**() method increments branch outcome counters. The analysis code uses class variables *branch* and *pc* because the current version of BIT does not allow passing more than one argument to the analysis methods at this time, a shortcoming that will be fixed in future work.

```
import BIT.*;
public class BranchPrediction {
static DataOutputStream data_out = null;
static Hashtable branch = null;
static int pc = 0;

  public static void main(String argv[]) {
    String infilename = new String(argv[0]);
    String outfilename = new String(argv[1]);
    ClassInfo ci = newClassInfo(infilename);
    Vector routines = cigetRoutines();
    for (Enumeration e=routines.elements();e.hasMoreElements(); ){
      Routine routine = (Routine) e.nextElement();
      Vector instructions = routinegetInstructions();
      for (Enumeration b = routinegetBasicBlocks().elements(); b.hasMoreElements(); ) {
        BasicBlock bb = (BasicBlock) b.nextElement();
        Instruction instr = (Instruction)instructions.elementAt(bbgetEndAddress());
        short instr_type = InstructionTable.InstructionTypeTable[instrgetOpcode()];
        if  (instr_type == InstructionTable.CONDITIONAL_INSTRUCTION) {
          instraddBefore("BranchPrediction", "Offset", new Integer(instrgetOrigOffset()));
          instraddBefore("BranchPrediction", "Branch", new String("BranchOutcome"));
        }
      }
    String method = new String(routinegetMethod());
    routineaddBefore("BranchPrediction", "EnterMethod", method);
    routineaddAfter("BranchPrediction", "LeaveMethod", method);
    }
    ci.write(outfilename);
  }
}
```

Figure 2. Instrumentation Code: Branch Counting Tool

```
public class BranchPrediction {
  static Hashtable branch = null;
  static int pc = 0;

  public static void EnterMethod(String s) {
    System.out.println("method: " + s);
    branch = new Hashtable();
  }

  public static void LeaveMethod(String s) {
    System.out.println("stat for method: " + s);
```

```
        for (Enumeration e = branch.keys(); e.hasMoreElements(); ) {
            Integer key = (Integer) e.nextElement();
            Branch b = (Branch) branch.get(key);
            int total = b.taken + b.not_taken;
            System.out.print("PC: " + key);
            System.out.print("\t\ttaken: " + b.taken + " (" + b.taken*100/total + "%)");
            System.out.println("\t\tnot taken: " + b.not_taken + " (" + b.not_taken*100/total + "%)");
        }
    }

    public static void Offset(int offset) {
        pc = offset;
    }

    public static void Branch(int brOutcome) {
        Integer n = new Integer(pc);
        Branch b = (Branch) branch.get(n);
        if (b == null)
            b = new Branch();
        if (brOutcome == 0)
            b.taken++;
        else
            b.not_taken++;
    }
}
```

Figure 3. Analysis Code: Branch Counting Tool

Analysis code is likely to vary between different customized tools depending on what their functional requirements are. However, most of the instrumentation code presented in Figure 2 is likely to be the same for different customized tools since all of them will require some sort of navigation through different constructs within a class file. For instance, to dynamically count the number of instructions that get executed in a user program, we would only have to change the body of the loop that obtains different basic blocks as shown in Figure 4. Instead of checking whether the last instruction in a basic block is a conditional or not, we count the number of instructions present in a basic block to obtain the total instruction count.

```
for (Enumeration b = routine.getBasicBlocks().elements();b.hasMoreElements(); ) {
    BasicBlock bb = (BasicBlock) b.nextElement();
    bb.addBefore("ICount", "count", new Integer(bb.size()));
}
```

Figure 4. Changes Required to Instrumentation Code to Count Instructions instead of Conditionals

4. BIT Implementation

There were several different approaches that could have been taken to observe and measure the dynamic behavior of Java bytecodes. One possible approach would be to modify the JVM to produce relevant outputs. An advantage of this approach is that it is easier to obtain certain kinds of information that would either be difficult or impossible to obtain otherwise. For example, measurements of the JVM garbage collection implementation would require JVM modifications. A drawback of this approach is that each time we wanted to create a customized tool, we would have had revisit the JVM source to add or remove tracing code. An even bigger problem would be that even if we modified the JVM source, redistributing the tools would be difficult due to licensing restrictions. Furthermore, tracking changes made to the JVM implementation, as new releases became available, would also be difficult.

We originally considered using EEL as a basis for our design, but we concluded that EEL supported more functionality than we required. EEL was designed for the SPARC architecture, which is more complex than the JVM, and it proved to be overkill for instrumenting JVM bytecodes. Furthermore, developing the BIT system in Java allowed us to create highly portable instrumentation tools. Although EEL's object-oriented design was still followed, we modeled BIT's functionality after that of ATOM (i.e., only allowing executable instrumenting and not arbitrary editing), which we felt was still highly valuable and much easier to design, implement, and use.

4.1. Adding Method Calls

Since instrumentation process requires adding new method calls to a class file, class and method name as well as other constants about these methods need to be inserted to the *constant pool table*. The *constant pool table* is a place where different string constants, class names, field names, and other constants are stored for each class file. To support the ability to add a method call before or after a certain entity (e.g., method, basic block, etc.), the descriptor, the class name, and the method name of the method being inserted need to be present in the *constant pool table* of the code being instrumented. The *constant pool table* is also used as a place to store arguments to the analysis methods.

If the string "BranchOutcome" is used as an argument to the analysis methods, then it is interpreted as a special directive for obtaining the outcome of the branch instruction since there is no way of knowing what the outcome of the branch would be at instrumentation time. In this case, appropriate bytecode instructions are added to obtain the outcome of the branch at run-time and pass it to the analysis method.

There are several JVM method invocation instructions such as *invokestatic*, *invokevirtual*, *invokespecial*, and *invokeinterface*. In BIT, analysis method calls are inserted by using the *invokestatic* bytecode instruction and therefore, analysis methods have to be static. This also implies that objects cannot be associated with these methods. The *invokevirtual* bytecode could have been used, but to keep things simple, only the *invokestatic* instruction is used. To use *invokevirtual*, more complex sequences of bytecodes would have to be inserted in the instrumented program because an instance of the class would need to be created and manipulated.

5. Performance Results

In this section, we present results based on applying two example tools implemented using BIT to five Java applications: a benchmark suite, a lexical analyzer generator, a Java compiler, an LALR parser generator, and BIT itself.

We wrote several small tools, such as a branch counting tool and a dynamic instruction counting tool, to exercise BIT. The branch counting tool is a version of the tool presented in Figures 2 and 3, modified to produce less output. We built the dynamic instruction counting tool by inserting calls to analysis methods before basic blocks. The analysis method receives the sizes of the basic blocks and adds and prints them to show how many JVM instructions were executed during the course of the user program's execution. For the results presented here, we instrumented only the application code (i.e., not the Java library classes). Had we instrumented the library classes as well, the overheads would undoubtedly be higher.

To learn about the performance of BIT, three characteristics were measured on an Intel Pentium 200Mhz machine with 24 MB of memory running Microsoft Windows 95 and Sun Microsystems Inc.'s Java Development Kit (JDK) version 1.1.4. The characteristics measured were time required to instrument user programs, execution time of the instrumented programs, and the size of instrumented programs. For these measurements, Jmark 1.2.1 [8], a JVM benchmark suite from Ziff-Davis publishing company; JLex [3], a lexical analyzer generator for Java; EspressoGrinder [24], a Java compiler; CUP [16], a parser generator; and BIT were used as user applications on which the custom tools mentioned above were run. Jmark consists of 19 class files and benchmarks 11 different areas of Java performance, JLex consists of 23 class files, EspressoGrinder is composed of 105 class files, CUP consists of 40 class files, and BIT consists of 43 class files.

Table 1 summarizes the time taken for each tool to build the instrumented programs. As shown in Table 1, the time taken to instrument each of the five applications was under four minutes for both of the customized tools. The average time taken to instrument a single class file was about two seconds. EspressoGrinder has more classes and a larger code size, and this explains why it took more time to instrument EspressoGrinder than the other four applications. As native code compilers become available and we are able to compile BIT, the time required to instrument applications should decrease significantly. The first column in this table is the time required to compile Java files using *javac* compiler and is included to aid the reader in estimating cost of instrumenting class files relative to complation.. In the cases where we did not have the program sources, the compilation time is indicated as N/A.

Table 1. Time Required to Instrument User Programs

Application	Compilation Time	Branch Count	Dynamic Instruction
Jmark	N/A	14 seconds	17 seconds
JLex	15 seconds	89 seconds	89 seconds
EspressoGrinder	N/A	233 seconds	174 seconds
CUP	24 seconds	55 seconds	88 seconds
BIT	15 seconds	32 seconds	31 seconds

Table 2 shows the execution time of the instrumented programs for each tool.

The increase in execution time of the instrumented programs ranged from 23% to 150%. This increase is due to method invocation overhead when invoking analysis methods and the time actually spent in the analysis methods. For most of the programs, the execution time was increased by approximately a factor of one to one and a half. The lower overhead observed in Jmark is most likely due to its extensive use of the Java libraries (e.g., for the purpose of benchmarking graphics, etc.), which were not instrumented in this study.

Table 2. Execution Time of Instrumented User Programs

Application	Uninstrumented	BRANCH COUNT (raw/% increase over uninstrumented)	DYNAMIC INSTRUCTION (raw / % increase over uninstrumented)
Jmark	315 seconds	409 seconds / 30%	387 seconds / 23%
JLex	16 seconds	31 seconds / 94%	20 seconds / 25%
EspressoGrinder	6 seconds	15 seconds / 150%	12 seconds / 100%
CUP	7 seconds	14 seconds / 100%	8 seconds / 14%
BIT	89 seconds	183 seconds / 106%	167 seconds / 88%

To get a better understanding of the programs we instrumented, we also measured some of the basic dynamic characteristics of the programs. In particular, we looked at the average bytecode instructions executed per basic block. This average (computed dynamically) was measured to be 4.4 in JLex, 5.8 in EspressoGrinder, 3.2 in CUP, and 3.7 in BIT.

We also measured the dynamic frequency of conditional branch instructions in JLex, EspressoGrinder, CUP, and BIT, and observed that 11.4% of all instructions were conditional branches in JLex; 10.3% in EspressoGrinder; 12.9% in CUP; and 6.8% in

BIT.

BIT instrumentation caused an increase in the program size as well, and the overhead ranged from 14% to 37% as shown in Table 3. This increase in size is explained by the addition of entries in the constant pool table, which includes static arguments to the analysis routines, the names of class and analysis methods, and method descriptors, and by the addition of actual bytecodes to invoke the analysis routines. The size of the program instrumented with the dynamic instruction counting tool increased more than that of the program instrumented with the branch counting tool except for one application. This is because the former includes bytecodes to invoke the analysis routines before each basic block while the latter only includes bytecodes to invoke the analysis routine before conditional instructions. However, since EspressoGrinder has a relatively large number of instructions per basic block compared to the other applications, the branch counting tool had more overhead in this application.

Table 3. Code Size of Instrumented User Programs

Application	Uninstrumented	BRANCH COUNT (raw / % increase over uninstrumented)	DYNAMIC INSTRUCTION (raw / % increase over uninstrumented)
Jmark	34,966 bytes	44,130 bytes / 26%	47,854 bytes / 37%
JLex	86,350 bytes	107,869 bytes / 25%	111,,173 bytes / 29%
EspressoGrinder	295,281 bytes	379,879 bytes / 29%	374,,460 bytes / 27%
CUP	117,964 bytes	147,738 bytes / 25%	151,933 / 29%
BIT	95,680 bytes	108,404 bytes / 13%	110,953 bytes / 16%

The results presented show that the prototype version of BIT has acceptable performance for several instrumentation tasks. We anticipate that the overheads of the prototype can be reduced dramatically by making a number of performance enhancements. To reduce the time required to instrument user programs, we could use arrays instead of vectors since vectors in Java tend to be two to three times slower than arrays according to our personal experiences. To increase the execution speed of the instrumented user programs, we could inline analysis methods, removing method invocation overhead (see [30]). Allowing multiple arguments to analysis routines would significantly decrease the number of method calls required to do the instrumentation, and would also substantially improve performance.

6. Summary

BIT is a set of interfaces that brings the functionality of ATOM and other related tracing tools to the Java world by allowing a user to instrument a JVM class file. Being able to create customized tools to observe and measure the run-time behavior of programs is valuable for many tasks including program optimization and system design.

BIT is the first framework that allows users to create customized tools to analyze JVM bytecodes quickly and easily. BIT allows the user to add calls to analysis methods any place in the JVM bytecode to obtain dynamic information about the program. We conducted a performance study based on two small tools written using BIT, and we reported the results here. The overheads for both the code size and the execution time were between 23% to 150%.

There are issues that need to be addressed in the near future, and in the rest of this section, we discuss them. One issue is the handling of exceptions. An exception in Java bytecode contains information about the exception handler in the code buffer, but since BIT changes the code buffer as a result of adding method calls, the information about the exception handler in an exception is no longer valid. This could result in run-time errors since incorrect exception handlers could be invoked when exceptions are raised. Checks are needed to make sure that the information about the exception handlers are also updated if they are going to be affected by changes in the code buffer. Exceptions are being ignored in the current implementation.

Larger customized tools using BIT need to be built to prove BIT's usefulness. Possible candidates includes a tool that performs hierarchical profiling of a class file such as gprof [13], HiProf [10], and mprof [34]. Recently, there have been other advances in profiling including flow and context sensitive profiling [2] and interprocedural dataflow analysis [11]. These advanced profiling techniques could also be applied to JVM programs with BIT.

7. Availability

BIT source is freely available. If you would like to obtain a copy, please emailhanlee@cs.colorado.edu or zorn@cs.colorado.edu

Acknowledgments

This work was supported in part by the National Science Foundation under grant IRI-95-21046. We would like to thank Margo Seltzer and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] Anant Agarwal, Richard L. Sites and Mark Horowitz. "ATUM: A New Technique for Capturing Address Traces Using Microcode." *Proceedings of the 13th International Symposium on Computer Architecture*, pages 199-127, June 1986.
[2] Glenn Ammons, Thomas Ball, and James R. Larus. "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling." In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85-108, June 1997.
[3] Elliot Berk. JLex: A Lexical Analyzer Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/JLex.
[4] Brian Bershad et al. Etch Overview. http://www.cs.washington.edu/~bershad/Etch.html.
[5] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook and William E. Weihl. "PROTEUS: A High-Performance Parallel-Architecture Simulator." Massachusetts Institute of Technology technical report MIT/LCS/TR-516, 1991.
[6] Per Bothner. Kawa, the Java-based Scheme System. http://www.cygnus.com/~bothner/kawa.html.
[7] Robert F. Cmelik and David Keppel. "Shade: A Fast Instruction-Set Simulator for Execution Profiling *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128-137, May 1994.
[8] Richard V. Dragan and Larry Seltzer. Java Speed Trials *PC Magazine*, vol 15, no 18, 1996.
[9] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor." *SIGMETRICS Conference on Measurement and modeling of Computer Systems*, vol 8, no 1, May 1990.
[10] Janel Garvin. HiProf Advanced Code Performance Analysis Through Hierarchical Profiling. http://tracepoint.galatia.com/noframes/products/hiprof/profiling/overview.
[11] David W. Goodwin. "Interprocedural Dataflow Analysis in an Executable Optimizer." In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 122-145, June 1997.
[12] James Gosling, Bill Joy, and Guy Steele *The Java Language Specification*. Addison-Wesley, 1996.
[13] Susan L. Graham, Peter B. Kessler and Marshall K. McKusick. "An Execution Profiler for Modular Programs." Software Practice and Experience, pages 671-685, vol 13, 1983.
[14] Jonathan C. Hardwick and Jay Sipelstein. "Java as an Intermediate Language." Technical Report CMU-CS-96-161. Department of Computer Science. Carnegie Mellon University, August 1996.
[15] Reed Hastings and Bob Joyce. "Purify: Fast Detection of Memory Leaks and Access Errors *Proceedings of the Winter USENIX Conference*, Pages 125-136, January 1992.
[16] Scott Hudson. Java Based Constructor of Useful Parsers (CUP). http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html.
[17] Intermetrics. AppletMagic: Ada for Java Virtual Machine. http://www.appletmagic.com.
[18] James R. Larus and Thomas Ball. "Rewriting Executable Files to Measure Program Behavior *Software, Practice and Experience*, vol 24, no. 2, pages 197-218, February 1994.
[19] James R. Larus and Eric Schnarr. "EEL: Machine-Independent Executable Editing." In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291-300, June 1995.
[20] Han B. Lee. *BIT: Bytecode Instrumenting Tool*. MS thesis, University of Colorado, Boulder, CO, July 1997.
[21] Tim Lindholm and Frank Yellin *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[22] MIPS Computer Systems, Inc.*Assembly Language Programmer's Guide*, 1986.
[23] J. Eliot B. Moss and Antony L. Hosking. "Approaches to Adding Persistence to Java.*First International Workshop on Persistence and Java*, Septermber 1996.
[24] Martin Odersky, Michael Philippsen, and Christian Kemper. EspressoGrinder. http://wwwipd.ira.uka.de/~espresso/.
[25] ODI. *PSE/PSE Pro for Java API User Guide.* 1997.
[26] Srinivasan Parthasarathy, Michael Cierniak, and Wei Li. "NetProf: Network-based High-level Profiling of Java Bytecode." Technical Report 622, Computer Science Department, University of Rochester, May 1996.
[27] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers." In*Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48-60, May 1993.
[28] Michael D. Smith. "Tracing with Pixie." Memo from Center for Integrated Systems, Stanford Univ., April 1991.
[29] K. So et al. "PSIMUL – A System for Parallel Execution of Parallel Programs." in*Performance Evaluation of Supercomputers*, J.L. Martin, ed., Elsevier Science Publishers B.V., North Hoolan, 1988.
[30] Amitabh Srivastava and Alan Eustace. "ATOM A System for Building Customized Program Analysis Tools." In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI),* pages 196-205, June 1994.
[31] Amitabh Srivastava and David Wall. "Link-Time Optimization of Address Calculation on a 64-bit Architecture." In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49-60, June 1994.
[32] Amitabh Srivastava and David Wall. "A Practical System for Intermodule Code Optimization at Link-Time.*Journal of Programming Languages*, vol 1, no 1, pages 1-18, March 1993.
[33] David W. Wall. "Systems for Late Code modification." In Robert Giegerich and Susan L. Graham, eds.*Code Generation – Concepts, Tools, Techniques*, pages 275-293, Springer-Verlag, 1992.
[34] Benjamin Zorn and Paul Hilfinger. "A Memory Allocation Profiler for C and Lisp Programs.*USENIX Conference Proceedings*, pages 223-237, Summer 1988.

HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching

Fred Douglis*
AT&T Labs – Research

Antonio Haro†
College of Computing, Georgia Institute of Technology

Michael Rabinovich‡
AT&T Labs – Research

Abstract

A number of techniques are available for reducing latency and bandwidth requirements for resources on the World Wide Web, including caching, compression, and delta-encoding [12]. These approaches are limited: much data on the Web is dynamic, for which traditional caching is of limited use, and delta-encoding requires both a common version base against which to apply a delta and the complete generation of the resource prior to encoding it. In contrast to these approaches, we take an application-specific view, in which we separate the static and dynamic portions of a resource. The static portions (called the *template*) can then be cached, with (presumably small) dynamic portions obtained on each access. Our HTML extension, which we refer to as HPP (for **HTML Pre-P**rocessing) supports resources that contain variable number of static and dynamic elements, such as query responses.

Results with macro-encoding of query response resources from local CGI scripts and two popular search engines indicate that our approach promises a substantial reduction of network traffic, server load, and access latency for dynamic documents. The size of network transfers using HPP are comparable to delta-encoding (factors of 2–8 smaller than the original resource), while the data generated by content providers is simpler, and the load on the end-servers is slightly lower.

*Email: douglis@research.att.com.

†This work was done while the author was visiting AT&T Labs–Research. Email: haro@cc.gatech.edu.

‡Email: misha@research.att.com.

1 Introduction

Caching plays a crucial role in a wide-area distributed system such as the World Wide Web. It significantly reduces response time for accessing cached resources by eliminating long-haul transmission delays. It also reduces backbone traffic and the load on content-providers.

The importance of caching will increase dramatically once ISDN lines and cable modems replace slow modems as the "last link" to the user. Indeed, a slow link to the user currently serves as a "floodgate" that limits the rate of requests from this user. With an order-of-magnitude increase in bandwidth provided by ISDN and cable modems, these floodgates will open, shifting the bottleneck further to the backbone and content servers.

However, a significant portion of Web resources are not cacheable, either because the resource is modified upon every access, or because the content provider explicitly prohibits caching [4]. Thus, increasingly sophisticated caching techniques are applied to a decreasing portion of resources on the Web.

Some proposals have been made to transmit encodings of the changes between subsequent versions of a resource, in order to reduce bandwidth requirements and improve end-to-end performance [1, 8, 12, 14]. With delta-encoding, one might cache a resource even if it is considered uncacheable, but not present the cached data without first obtaining the changes to present its current version. One advantage of these proposals is that they apply uniformly to all uncacheable resources regardless of the reason why they are uncacheable. Another advantage is that they can be implemented transparently, via proxies, so that content providers need not be modified. However, the delta-encoding proposals have disadvantages as well. If the content provider must compute the delta-encodings on the fly, it suffers overhead and must store a potentially large number of past versions; if the delta computation is performed by an intermediary, then the entire resource must be sent from the content provider to the intermediary, and the encoding must still be performed on the "critical path."

We have observed that with a common class of resources, such as those provided by search engines, a significant part of the resource is essentially static. Portions of the resource vary to different extents from one response to another (the difference between two pages in response to a single query is usually smaller than the difference between pages from different queries). Also, the location of the dynamic portions relative to the rest of the resource does not change. Consider, for example, a document generated in response to a stock quote query. It contains the banner identifying the content provider, headers, information specifying formatting and fonts, and, finally, the name and the stock price of the requested company. The banner, headers, and formatting stay the same, and the dynamic portion (the name and the stock price) go in the same place within the resource.

We therefore have extended HTML to allow the explicit separation of static and dynamic portions of a resource. The static portion contains macro-instructions for inserting dynamic information. The static portion together with these instructions (the *template*) can be cached freely. The dynamic portion contains the bindings of macro-variables to strings specific to the given access. The bindings are obtained for every access, and the template is expanded by the client prior to rendering the document. In other words, a macro-preprocessing phase at the client permits *partial* caching of dynamic resources.

We designed our macro-encoding language, which we refer to as HPP (for **H**TML **P**re-**P**rocessing), to minimize the size of the bindings. We motivate our proposal by examples from several popular resources, all of which show a remarkable difference in size between the original resource and the bindings: factors of 4–8 without compression, or 2–4 when comparing compressed bindings to the compressed original resource.

In addition to gains in performance, HPP should make authoring dynamic resources easier. Instead of writing programs that generate the full HTML document, the bulk of the document can be generated using an HTML authoring tool similar to the ones available now, and only the dynamic portions will have to be produced as a program output. In fact, similar techniques are already used to simplify programming (the "shtml" server-side include feature of many HTTP servers). Our proposal allows a systematic and more general way of doing this, and also exploits this separation to enable caching. Macro-preprocessing is almost a client-side equivalent to server-side inclusion, but with a slightly more sophisticated language that supports, for instance, looping constructs.

The rest of the paper is organized as follows. We discuss the extension to HTML for client-side macro-preprocessing in Section 2. Section 3 outlines the implementation path of our approach within the HTTP protocol and HTML language. Section 4 evaluates the performance of our approach using several existing dynamic resources including two well-known search engines. A more detailed comparison with related work is given in Section 5. We conclude in Section 6 with a summary of main results.

2 HTML Extension

Our proposal for macro-preprocessing within HTML is similar to source code preprocessing (e.g., *cpp*). In our case, the template is first expanded into the actual HTML document, which is then rendered on the browser screen. The HTML extension contains the following new tags: VAR, LOOP, IF, SET_VAR, and DYNAMICS. VAR, LOOP, IF, and SET_VAR are used in the template to compose instructions regarding the location of dynamic content. DYNAMICS is used to delineate the dynamic part of the document. It contains bindings of the macro-variables to dynamic text strings specific to the current document access.

2.1 Overview

The syntax of the new tags conforms with the SGML specification [6]. We describe these tags informally using a series of examples, without spending time on straightforward details.

The VAR tag is used to include a macro-expression in the text. The operations defined in the macro-expression are concatenation, basic arithmetic operations (which obviously make sense only if the operands evaluate to numbers), etc.

In many cases, a macro-expression contains a single variable and is used in the template as a placeholder, to be replaced by a text segment dynamically bound to this variable. An example of this approach appears in Figure 1 with "time" and "count" variables.

As a more complicated example, consider the response from Lycos[1] [11] to the query "caching dynamic objects." At the end of the pageful of results, the response

[1]All fragments of Lycos output are Copyright ©1994-1997 Carnegie Mellon University. All rights reserved. Lycos is a trademark of Carnegie Mellon University. Used by permission.

```
<HEAD>
<TITLE>Michael Rabinovich</TITLE>
<BODY>
...
The time is <VAR time>.
This page has been accessed <VAR count> times.
</BODY>
```

(a) The template for a simple query.

```
<HTML>
<TEMPLATE HREF="query.hpp">
<DYNAMICS>
time = 1:15pm;
count = 10;
</DYNAMICS>
</HTML>
```

(b) The bindings for a particular instantiation.

Figure 1: Example of HTML preprocessing encoding scheme.

```
<FONT SIZE=+1>1</FONT>
&nbsp;.&nbsp;
<A HREF="/cgi-bin/pursuit?first=11&part=&cat=lycos&query=caching+dynamic+objects">2</A>
&nbsp;.&nbsp;
<A HREF="/cgi-bin/pursuit?first=21&part=&cat=lycos&query=caching+dynamic+objects">3</A>
<...>
```

Figure 2: Output from the sample Lycos query, with links to additional pages of results.

provides links to the next ten pages of the results. These links are generated by the HTML fragment in Figure 2. This fragment is rendered in the browser window as 1.2.3.4.5.6.7.8.9.10, where each number is associated with a query URL. The URL contains the keywords and the number of the first result of interest. This fragment could be encoded in our extended HTML as the template and bindings shown in Figure 3, which illustrates the use of a more complex VAR expression, as well as the simple use of a LOOP tag. All variables in the DYNAMICS portion of the resource can be specified explicitly, separated by commas, or as a numeric range similar to a FOR loop with range and increment.

To arrive at the actual fragment, the pre-processor would expand the template fragment within the loop as many times as the number of bindings to the subcount macro-variable specified in the loop fragment of the DYNAMICS section. Each time the expansion is done with the new binding for the subcount variable. However, the same binding for the query variable is used in each expansion because the binding is specified outside the LOOP. Notice that the amount of dynamic information that changes with each access is significantly smaller in our encoding.

The next example concerns conditional macro-expansion. Consider, again, the fragment of Lycos output showing the numbers of ten pages of results. The number of the currently viewed page is included as plain text, while other page numbers are part of the anchors referring to corresponding URLs as shown in Figure 2. For example, if the current page is 4, the fragment would be rendered as 1.2.3.4.5.6.7.8.9.10. This fragment could be macro-encoded by splitting the loop of Figure 3 into two, with the first loop generating page numbers preceding the current page, followed by the current page number in a VAR expression, followed by the second loop generating the remaining page numbers. Another complication is that there are ten numbers in the fragment but only nine dots in-between. So, without extra functionality, one would have to encode the first number separately outside the loops, since each loop iteration adds a dot and a number. Figure 4 illustrates a more convenient way of macro-encoding the same fragment using conditional statements and assignments to macro-variables. We have not fully implemented conditional statements and assignments yet. Consequently, we hard-coded the first page number in the template as in Figure 2, assuming that separate templates are prepared for the second, third, etc pagefuls of results. Note that our shortcut caused negligible decrease in the size of the template and bindings compared to the encoding with conditionals and assignments (0.85% decrease for the template and 0.35% decrease for the bindings), so it does not affect our performance conclusions.

Our final example illustrates a more complex use of a LOOP construct. Considering the same query to Lycos as before, Figure 5 shows the fragment of the response that generates a list of ten results (only the first and the

```
<FONT SIZE=+1>1</FONT>
<loop> &nbsp;.&nbsp;
<A HREF="/cgi-bin/pursuit?first=
<var 10*(subcount - 1)+1>&part=&cat=lycos&
query=<var +query>"><var subcount></A>
</loop>
```

(a) The template.

```
query = caching+dynamic+objects
<loop>
subcount = 2 to 10,1;
</loop>
```

(b) The bindings.

Figure 3: Example of HTML preprocessing for the fragment of the Lycos query output shown in Figure 2. The example demonstrates a LOOP construct. The template corresponds to the first portion of the query results - see Figure 4 for the template that applies universally to all portions.

```
<set_var loop_start=1>
<loop>
<if loop_start = 0>
   &nbsp;.&nbsp;
</if>
<set_var loop_start=0>
<if subcount = active>
   <FONT SIZE=+1><var subcount></FONT>
<else>
   <A HREF="/cgi-bin/pursuit?first=
      <var 10*(subcount - 1)+1>&part=&cat=lycos&
      query=<var query>"><var subcount></A>
</if>
</loop>
```

(a) The template.

```
query = caching+dynamic+objects;
active = 4;
<loop>
subcount = 1 to 10,1;
</loop>
```

(b) The bindings.

Figure 4: Example of conditional macro-expansion (the current page is 4).

last of the results are shown). The fragment can be obtained from the template and bindings shown in Figure 6.

2.2 Scoping

The scope of macro-variables is similar to scope in block-structured languages. Variables whose binding is specified within a loop in the DYNAMICS section have the scope limited to that loop. Variables that are bound outside any loop in the DYNAMICS section have global scope. They are shadowed by loop variables with the same name. Only a single binding can be specified to each global variable, and this binding is used everywhere this variable is encountered in the template. The order in which loop and global variables appear in the template loop is immaterial.

Loop variables have a list of values in their binding specifications. In an iteration, each variable is bound to the next value in its list, and the loop fragment of the template is expanded using these bindings. When the list is exhausted for some variable, the expansion stops, and the rest of all other lists is ignored.

The LOOP construct can be nested. This could be used to macro-encode a resource that contains a variable number of sections, each containing a variable number of similar entries. Consider, for example, a hypothetical service that gives restaurant information. A query could be: "Give me Chinese restaurants under $10 in Union County". The response lists results by a town. For each town, the resource provides general information (e.g., links to current events in the town or other categories of restaurants), and a list of Chinese restaurants. Thus, the resource contains a number of town entries, whose number varies with a query, with a variable number of restaurant entries within each town entry, again, depending on the query. One can easily provide examples of deeper loop nesting.

```
<FONT SIZE=-1 FACE=ARIAL, HELVETICA><b>1)
<a href="http://amsterdam.lcs.mit.edu/papers/www94.html"> Dynamic Documents </a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
    Dynamic Documents: Extensibility and Adaptability in the WWW
    M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber
    MIT Laboratory for Computer Science...</font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    http://amsterdam.lcs.mit.edu/papers/www94.html
    [100%, 3 of 3 terms]</font><p>

<...>

<FONT SIZE=-1 FACE=ARIAL, HELVETICA><b>10)
<a href="http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html">
    SunWorld Online distributed object glossary</a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
     [Table of contents][Sun's homepage][Next story][Sidebar][Back to story]
    Glossary of Object-Oriented Terminology for Business Compiled by Mike Aube an...</font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html
    [28%, 2 of 3 terms]</font><p>
```

Figure 5: Responses from the sample Lycos query.

When a template and the corresponding bindings contain multiple loops, some of which are nested, the correspondence between the loops in both parts is established structurally. For the top-level loops, the first loop in the template corresponds to the first loop in the bindings, etc. Within each top-level loop, the correspondence is established similarly for the next-level nested loops, and so on. It is an error to have non-matching loop structures of the template and bindings of the same resource.

One could provide further features of general macro-languages, like nested macros, where the binding to a variable contains a VAR-expression, or specifying full-blown macro-definitions, including macro-parameters, in the bindings. For example, when different resource representations are generated for different types of browsers (e.g., with or without support for frames) a single template can be cached at the proxy and conditionally expanded by different browsers without contacting the content provider.

3 HTTP and HTML Macro-expansion Implementation

While the previous section presented an overview of HTML macro-expansion, this section discusses issues involving HTTP and caching.

3.1 Methodology

The goal of HPP is to cache static content while permitting dynamic content to be transferred when needed. The browser would have to use an Accept-encoding request header to inform the server that an HPP encoding would be understood. In the steady state, a browser would have many commonly used templates in its cache, and receive a response header such as Content-encoding: x-hpp to describe dynamic data that has a reference to a template.

The question is, how should HPP handle a template that is *not* already cached? In the worst case, each x-hpp resource would refer to a template that is not already cached, and require a second round-trip to the content provider to obtain the template. Even with techniques such as the Keep-alive request header of HTTP 1.1 [5], which would use a single TCP connection to request both resources, the extra round-trip could dramatically increase the end-to-end latency to receive the original resource.

An alternative is to send the template along with the dynamic data as a MIME multipart document [7] when it is not cached already. This would require a mechanism for deciding when to send the template. One way to do this is to establish a one-to-one correspondence between a resource URL and an identifier (e.g., URL) of its template. This would let the client determine *a priori* if it has the template for a given URL in its cache, and then pass this information in an HTTP header, together with a version

```
<loop>
<FONT SIZE=-1 FACE=ARIAL HELVETICA><b><var counter>)
<a href="<var queryurl>"> <var querysubj></a>
</b></font><br><FONT size=-1 FACE=ARIAL, HELVETICA>
    <var querysum></font>
<br><FONT SIZE=-1 FACE=ARIAL, HELVETICA>
    <var queryurl>
     [<var percent>%, <var nummatch> of <var numterms> terms]</font><p>
</loop>
```

(a) The template corresponding to the fragment of Lycos response shown in Figure 5.

```
<loop>
counter = 1 to 10, 1;
queryurl=
    "http://amsterdam.lcs.mit.edu/papers/www94.html",
    <...>
    "http://fbp.icm.edu.pl/sunworldonline/swol-04-1996/swol-04-oobook.glossary.html";
querysubj=
    "Dynamic Documents",
    <...>
    "SunWorld Online distributed object glossary";
querysum=
    "Dynamic Documents: Extensibility and Adaptability in the WWW
    M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber
    MIT Laboratory for Computer Science...",
    <...>
    "[Table of contents][Sun's homepage][Next story] [Sidebar][Back to story]
    Glossary of Object-Oriented Terminology for Business Compiled by Mike Aube an...";
percent= 100, <...>, 28;
nummatch= 3, <...>, 2;
</loop>
```

(b) The bindings.

Figure 6: Example of a more complicated LOOP construct, with multiple explicit values per variable.

identifier of the cached template. (A version identifier, like a last-modified timestamp or an *etag* [5] is needed to ensure that the cached template is still current.) The downside of this approach is that it would reduce the flexibility of HPP. For instance, multiple resources might share a single template in the absence of this restriction. More importantly, some content providers generate significantly different output for the same URL, e.g., depending on the type of requesting browser. Thus, a URL may correspond to multiple templates.

The simplest way to address the above problems, and the one we take initially, is to include explicit template URL in bindings. The server first returns the bindings with the proper template URL, which is then fetched by the client unless cached. This approach assumes that templates will commonly be cached and that the added cost of a second request when a template is not already cached will be amortized over the benefits that otherwise accrue. A recently published study of Web accesses indirectly supports this assumption, showing an 85-87%

rate of repeated access to dynamic resources (see [12], Sections 5.1 and 5.2). If HPP reaches a wide enough distribution to gather meaningful statistics of real usage, it will be possible to determine how valid this assumption is in practice.

3.2 Comparison with other Techniques

HPP serves two purposes: to allow the dynamic portions of similar resources to be transferred without sending the static portions, and to allow a compact representation of repetition within a resource (through the use of the LOOP construct). In fact, both of these goals can be achieved through other means: the former via delta-encoding [1, 8, 12, 14] and the latter via compression [12, 13]. A fundamental issue with delta-encoding is the management of past versions, since a server and client must agree on a base version against which to apply a delta. The server may generate thousands of versions of a single dynamic resource and cannot easily store every past version that an arbitrary client might have stored. HPP addresses this problem by permitting a single cached template that all clients may cache, and providing additional dynamic data in the context of that template. A delta-encoding system could similarly notify clients that they should store a particular instance of a resource, against which future deltas would be computed [8]. Finally, delta-encoding does not help with repetition unless the encoding itself is compressed [12].

Simple compression of responses would eliminate redundancy from loops in a manner similar to HPP. In fact, compression could potentially achieve a better reduction in data size because it would compress all text, not just the obvious repetition from loops. However, as shown below, HPP templates and dynamic data can be compressed as well, getting the benefits of both compression and caching of the static portions.

Part of the rationale behind HPP is that it provides servers with the opportunity to generate *just* the dynamic data, rather than generating an entire HTML resource only to compute a delta-encoding or compressed version of it. The expected data transfer size in either case is comparable, but the server overhead should be less. Section 4 discusses performance issues.

3.3 Compatibility

HPP requires extra functionality from the client (to perform macro-preprocessing of templates) and the server (to recognize the x-hpp header from clients willing to accept the HPP encoding). Extra functionality can be added to clients without modifying existing browsers by co-locating a proxy with the browser [2]. CGI scripts on servers will have to understand how to generate HPP dynamic data, but that process can be automated via libraries or other tools.

HPP could also be implemented with no modification of client software using Java applets or as a plug-in. In the Java approach, when a client requests a dynamic resource, a page is returned containing the URL of a Java applet that implements HPP expansion and, as the parameters to the applet, the URL of the template and the bindings of the resource. The client would then fetch the applet (presumably, it would be cached in most cases since it is the same for all resources) and pass it the bindings and the template. The applet would then fetch the template and perform the expansion.

4 Performance Evaluation

To quantify the potential benefits of HPP, we have used it to encode dynamic resources in several contexts. Most of our experiments have been performed using a modified version of an internal Web-based "recruiting database." We chose this application for two reasons. First, it typifies the style of response that is well-suited to HPP: a query returns just a list of names that are hyperlinks to CGI invocations with the name of the candidate specified, or else returns a full display of all information about one or more candidates in a canonical form. Second, unlike search engines and other services both on the external Internet and elsewhere within the AT&T Intranet, we have full access to the CGI scripts, which we have modified to use HPP. In addition, to show broader applicability of HPP, we have encoded by hand some other dynamic resources, as described in Section 4.2.

4.1 Metrics

Useful metrics for evaluating HPP include user request latency, network bandwidth demands, and the load placed on content providers. We anticipate that templates will be generated as part of application development, using authoring tools similar to the ones used today for creating static documents. We assume that templates will be cached aggressively by clients; therefore, our evaluation focuses on the dynamic aspects of a query. On the server, we consider the overhead of generating the dynamic portion of a resource, compared to generating the entire resource as in traditional systems and then optionally computing a delta-encoded or compressed form. On the network, the total number of bytes transferred is an issue, though for small resources the round-trip time will dominate any bandwidth issues (i.e., reducing a 500-byte resource to a 200-byte resource will have minimal effect, but reducing a 5-Kbyte resource to a 2-Kbyte resource will be more noticeable). On the client side, reconstructing the original resource from HPP, a delta-encoded response, or a compressed response will add overhead in comparison to simply displaying an unencoded resource.

Mogul et al. [12] measured the performance of delta-encoding and compression on both the client and server, and found that a library-based encoding and decoding system (one that could be linked into an HTTP client or server, rather than invoked as a separate process) is significantly faster than a T1 line (193 KBytes/sec). For HPP, the combination of overhead and reduced bandwidth needs to be competitive with these other encodings for it to be practical. Our thesis is that by permitting an application to generate the dynamic data without dealing with the portions of the HTML resource that do not change between invocations, HPP encoding can be more efficient than generating the resource and then delta-encoding or compressing it. Also, the templates and bindings can themselves be compressed efficiently, for further size reductions.

Our experiments were conducted using Apache Web server running on a 200MHz Sun Ultra 1 workstation with 128 Mbytes of memory. For comparison with compression and delta-encoding, we used *vdelta*, the fastest known system for compression and delta-encoding [9].

4.2 Sample Data

In our preliminary experimental evaluation of HPP, we consider five HTML resources:

A The output of a query to the modified recruiting database (mentioned at the start of this section), listing 10 candidates by name only.

B The same query, listing candidates with full information.

C The same query as **B** but listing only one candidate.

D, E Queries to AltaVista [3] and Lycos [11] using the query string "caching dynamic objects."

4.3 Bandwidth Demands

Figure 7 shows the size of each resource using several formats: the unencoded resource, compressed using *vdelta*, delta-encoded using *vdelta*, and using HPP with and without *vdelta* compression. For the delta-encoding, we considered several possible base versions: a substantially similar resource, such as the first page of the response to a search engine query compared with the second page; a somewhat different response from the same search engine (searching for a different set of keywords); and the HPP template, which has much of the text that appears in the final resource. Here we present delta-encodings against the template, under the assumption that the template is similar to what a system like WebExpress [8] might do in designating a base version. Finally, in the case of HPP, the bars are broken into two components, with the lower bar indicating the dynamic resource and the higher one the template, which would presumably frequently be cached.

Purely from the standpoint of network bandwidth, when templates are cached, the size of the compressed HPP dynamic data is comparable to an efficient delta-encoding. The uncompressed dynamic data, though larger, is still significantly smaller than even the compressed original resource.

4.4 End-To-End Latency

Figure 8 shows end-to-end request latency using each type of encoding. For resources **A** through **C**, the figure reports actual latencies measured over 28.8K modem and averaged over 100 requests. Since we did not have a good implementation of the Web server that would incorporate compression and delta-encoding, the experiments with these encodings were conducted using pre-computed compressed and delta-encoded resources, so that the server would output the generated resource to /dev/null and read the appropriately pre-computed resource from a file to return to the client. Given that the overhead for encoding and decoding resources with *vdelta* is negligible (this overhead, shown later in Table 2, is always below 0.5% of the latency), it is omitted. We do include the HPP expansion overhead on the client, which ranges from 3 to 14% of the latency in our experiments.

For resources **D** and **E**, Figure 8 includes estimated transfer costs, based on the size of each resource, as well as measured HPP expansion costs from Table 2, using the following rationale. The transfer costs are estimated by using a fixed cost of 700ms for a connection set-up and a variable cost assuming the resource is transferred at 22Kbps. These parameters closely approximate measured latency for resources **A** and **B**. The measured latency for resource **C** was unexpectedly high; we are still investigating the reason for that. Again, these graphs do not include encoding and decoding costs for *vdelta*, because they would not be discernible. We also omit the overhead for generating the original resource and HPP dynamics because, first, measurements with the recruiting database scripts show that generating just the dynamic data is consistently slightly faster than generating the entire HTML resource, and second, AltaVista and

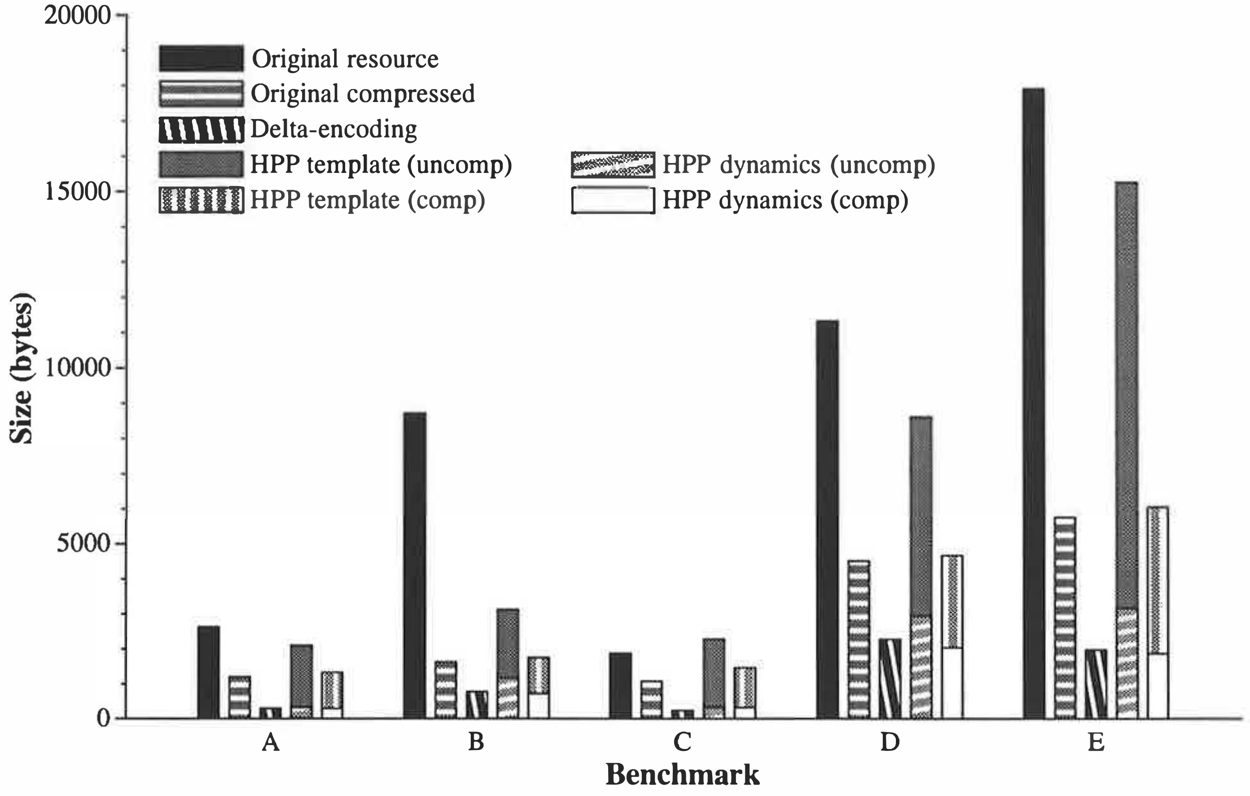


Figure 7: Sizes of the original and encoded resources for each query studied.

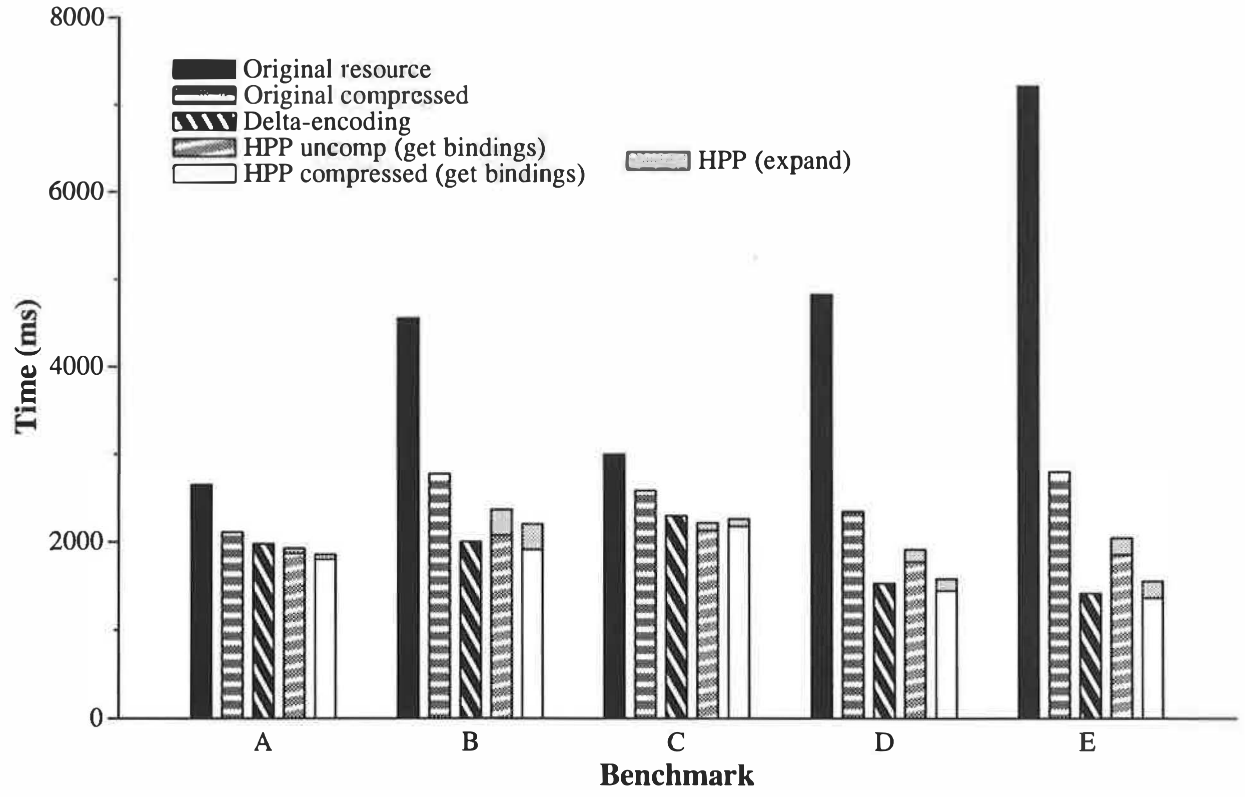


Figure 8: Measured or estimated end-to-end latency of the original and encoded resources for each query studied.

Resource	Full HML	HPP Dynamics
A	923.6ms	841.2ms
B	938.6ms	901.5ms
C	884.9ms	878.5ms

Table 1: Resource generation times (in ms) at the server.

Resource	Compress	Deltas	HPP expand
A	3.8	3.7	24.0
B	4.6	4.6	87.3
C	3.6	3.7	19.2
D	6.6	6.5	137.4
E	8.6	8.2	189.9

Table 2: Overheads for compression, delta-encoding, and HPP expansion (in ms).

Lycos are generated by extremely powerful machines so the latency should be dominated by a transmission over slow link. Also, without access to the Lycos and AltaVista CGI applications, we can only generate the dynamic data for these resources by hand.

Our measurements indicate that over a slow link, both delta-encoding and HPP can substantially reduce the end-to-end latency and overall network load. Compared to fetching uncompressed original resource, sending uncompressed HPP reduces the latency by between 26% for resource **C** and 72% for resource **D**, including the unoptimized macro-expansion on the client. For compressed resources, the latency reduction ranges from 12% to 44%, again including the expansion.

Comparing HPP and delta-encoding, they show essentially the same overall latency when taking into account HPP expansion on the client. If this expansion could be optimized to the level of *vdelta* overhead, HPP would result in between 4 and 9% less latency.

4.5 Server Load

Table 1 compares the time to generate the entire resource and just HPP dynamics. We could not include AltaVista and Lycos for resource and HPP generation without access to these applications. The results are averaged over 1000 invocations.

Table 2 gives the overhead for compressing original resources, computing *vdelta* with the template as the base version, and the HPP expansion time on the client. We timed *vdelta* and HPP encoding and decoding as the average of 100 loops within a single process that performed the operation; this method amortizes process startup costs and produces overhead comparable to a library implementation of an encoding and decoding system [12]. In addition, we have implemented the HPP expansion code as a "coprocess," again in order to amortize startup costs but to run in the context of Perl, with similar performance.

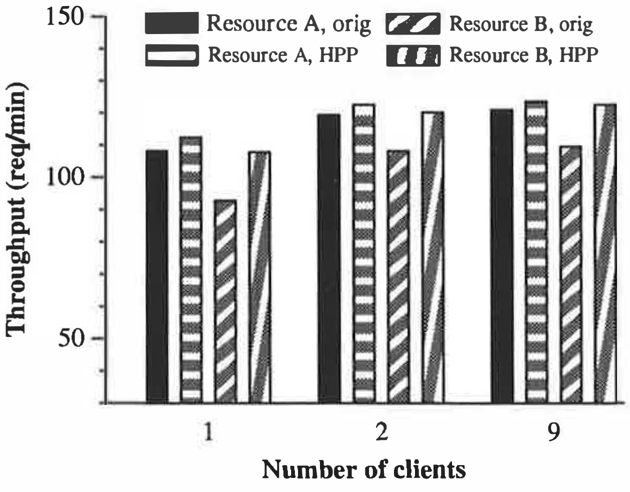


Figure 9: Measured server throughput.

Table 1 shows that generating just the dynamic data is slightly but consistently faster than generating the entire resource. Computing and applying *vdelta* is always negligible compared to resource generation. The overhead for decoding compressed or delta-encoded resources on the client (not shown) were even slightly lower than the encoding costs. However, HPP decoding on the client is discernible and is an order of magnitude higher than *vdelta* overhead. One should note that HPP expansion is implemented as unoptimized Perl script. In a real implementation, HPP expansion overhead should be comparable to applying a delta-encoded resource against a previous version. Even if it is more expensive computationally than applying a delta, HPP would have the benefit of shifting load from servers to clients.

The lower time to generate HPP dynamics, compared to the entire resource, translates into increased server capacity, as shown in Figure 9. This figure compares throughput of the server using full HTML and HPP. In this experiment, client machines run multiple processes, with each process repeatedly sending a request to the server, waiting for the response, and immediately sending the next request. Client machines are connected to the server via 10Mbps Ethernet. The reported datapoints include one client machine running one process, two machines running one process each, and three machines with three processes per machine. The throughput is measured as the number of processed requests per minute in a five-minute experiment.

The results show that HPP consistently increases server throughput, by up to 15%. While scripts for the recruiting database are implemented with an unoptimized Perl library so it is difficult to generalize these results, they indicate that HPP may provide a sizable improvement in server capacity.

5 Related Work

There are several existing approaches that could be used to improve performance of access to dynamic pages. In the *optimistic deltas* approach of [1], a proxy cache optimistically transfers to the client a document that may be out-of-date, while obtaining the current version from the content provider. Upon obtaining the current version, the proxy sends to the client a (possibly empty) delta from the old version to the new version. If the client already has some stale version, it can include its version ID (e.g, an etag [5]) with its request, and if the proxy also has this version, it can send the delta only.

The optimistic deltas target mostly static pages when they expire in caches or are modified by the authors. Applying this mechanism to dynamic pages that are different for every access would require the content provider (or the proxy) to keep a large version pool, which would be used to compute the delta against the requesting client's version. We view optimistic deltas as complimentary to our approach: deltas could be used to improve access to templates when they are modified or expire.

The WebExpress system [8] allows the content provider to specify a base version of a dynamic document and then send delta-encodings against this version. It thus avoids the need for a version pool, since the end server always computes the delta-encoding against the base version. The advantage of WebExpress over HPP is that is does not require any changes to HTML. HPP, however, offers equivalent or better performance by avoiding the computation of the full HTML resource before computing a delta-encoding.

The W3C consortium has proposed an extension for HTML that would allow embedding arbitrary resources into the HTML resource [10]. A similar extension has been proposed by Netscape as well. A new OBJECT tag is introduced, which can be used to specify the URL of resources to be inserted as well as the URL of the code to interpret these resources. The code specification may be implicit based on the resource type. This mechanism could in principle be used to insert dynamic elements of the resource into a static template, as in HPP. However, each dynamic element must be treated separately - each must be requested, and the executable must be invoked each time to insert the next element into the resource. In addition, this method does not supports loops. The whole loop fragment has to be treated as a dynamic element. These loop elements constitute a large portion of typical resources, especially those returned by search engines. Thus, much of the benefits of HPP would not be realized.

6 Conclusion

We have proposed using macro-preprocessing at the client to support caching of dynamic HTML documents. In our approach, called HPP, HTML is extended to allow separation of the static and dynamic portions of a resource. The static portions (called the *template*) can then be cached, and only dynamic portions must be obtained on every access.

We described our HTML extension informally using real resources as examples. We also showed on the example of two popular search engines that our macro-encoding results in a remarkable reduction of the amount of data that must be fetched from the server on every access. The dynamic data is comparable in size to an efficient delta-encoding, especially if the dynamic data is itself compressed. Compared to the original resource, HPP bindings reduce the size by factors of 4–8 without compression, or 2–4 when comparing compressed bindings to the compressed original resource. The load of the server is also decreased since it does not have to transmit the template for clients that already have it in their caches. In fact, when data compression is used, the server load should decrease even for requests that miss in the client caches: templates, which constitute a large portion of resources, can be compressed once and served repeatedly at low cost. Given that data compression dramatically reduces network traffic and delays, this can become an important factor in the future.

HPP requires minimal changes to Web servers and no changes to the HTTP protocol. The changes on the server side are concentrated in CGI scripts and other applications that generate dynamic data. The changes on the client side can be provided in various ways, including a custom proxy co-located with the browser, a plug-in, a Java applet, or direct support within the browser.

Recent trace data [12] indicate that dynamic resources exhibit a rate of repeated access that is more than twice the rate of repeated access to static resources. Therefore, they promise a much higher hit ratio if cached. Allowing caching of these resources should result in disproportional gains in overall Web access performance.

Acknowledgments

Gideon Glass and Balachander Krishnamurthy provided helpful comments on earlier drafts. We also thank the

anonymous referees for their comments.

References

[1] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of 1997 USENIX Technical Conference*, pages 289–303, Anaheim, CA, January 1997. Also available as http://www.research.att.com/~douglis/papers/optdel.ps.gz.

[2] Charles Brooks, Murray S. Mazer, Scott Meeks, and Jim Miller. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International WWW Conference*, December 1995. Also available as http://www.w3.org/pub/Conferen=-ces/WWW4/Papers/56/.

[3] Digital Equipment Corporation. http://www.altavista.digital.com, January 1997.

[4] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proceedings of the Symposium on Internetworking Systems and Technologies*. USENIX, December 1997. To appear.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, et al. RFC 2068: Hypertext transfer protocol — HTTP/1.1, January 1997.

[6] International Organization for Standardization. Standard generalized markup language, 1986.

[7] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part one: Format of Internet message bodies, December 1996.

[8] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM. Also available as http://www.networking.ibm.com/artour/artwewp.htm.

[9] James. J. Hunt, Kiem-Phong Vo, and Walter. F. Tichy. An empirical study of delta algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.

[10] Inserting objects into HTML. http://www.w3.org/pub/WWW/TR/WD-object.html.

[11] Lycos. http://www.lycos.com/, January 1997.

[12] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of SIGCOMM'97*, pages 181–194, Cannes, France, September 1997. ACM. An extended version appears as Digital Equipment Corporation Western Research Lab TR 97/4, July, 1997, available as http://www.research.digital.com/wrl/techreports/abstracts/97.4.html.

[13] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, andPNG. In *Proceedings ofSIGCOMM'97*, pages 155–166, Cannes, France, September 1997. ACM.

[14] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of SIGCOMM'96*, volume 26,4, pages 293–305, New York, August 1996. ACM. Also available as http://ei.cs.vt.edu/~succeed/96WAASF1/.

Lightweight Security Primitives for E-Commerce

Yossi Matias Alain Mayer Avi Silberschatz
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{matias, alain, avi}@bell-labs.com

Abstract

Emerging applications in electronic commerce often involve very low-cost transactions, which execute in the context of ongoing, extended client-server relationships. For example, consider a web-site (server) which offers repeated authenticated *personalized stock quotes to each of its subscribers (clients). The value of a single transaction (e.g., delivery of a web-page with a customized set of quotes) does not warrant the cost of executing a handshake and key distribution protocol. Also, a client might not always use the same machine during such an extended relationship (e.g., a PC at home, a laptop on a trip). Typical* transport/session-layer *security mechanisms such as SSL and S-HTTP either require handshake/key distribution for each transaction or do not support client mobility.*

We propose a new security framework for extended relationships between clients and servers, based on persistent shared keys. We argue that this is a preferred model for inexpensive transactions executing within extended relationships. Our main contribution is the design and implementation of a set of lightweight application-layer primitives, *for (1) generating and maintaining persistent shared keys* without *requiring a client to store any information between transactions and (2) securing a wide range of web-transactions (e.g., subscription, authenticated and/or private delivery of information, receipts) with adequate computational cost. Our protocols require public key infrastructure only for servers/vendors, and its usage only once per client (upon first interaction).*

1 Introduction

Considerable attention has been given recently to *transport/session-layer* security mechanisms. There are several proposals and implementations available, including SSL [SSL96], S-HTTP [SHTTP95], and SSH [SSH96]. Offering security mechanisms at the transport/session layer has the advantage of obtaining universal security primitives which have wide applicability (e.g., SSL or SSH can be used in conjunction with any TCP connection). However, universality in the proposed schemes comes at the expense of lacking flexibility with respect to complexity and cost of securing transactions, which vary in terms of their monetary value. In particular, web-transactions within the same client-server relationship but executing at different times, either appear unrelated to the transport layer or require the client to store data in secure memory, thus putting additional responsibility on the client and preventing mobility.

Emerging applications in electronic commerce often involve very low-cost transactions, which execute in the context of an ongoing, extended client-server relationship. Rivest predicts in [R97] the increase of low-cost transactions and the need for "low-cost crypto". For such transactions, general-purpose security mechanisms tend to be prohibitively expensive. In particular, both SSL and S-HTTP involve handshake/key-distribution that consist of a costly public key cryptography. We argue that a framework based on a *shared key* between a client and a server, *persistent* for the whole duration of a relationship, is an attractive choice. From a technical point of view, the main challenge is in obtaining low-cost establishment and maintenance of the persistent shared keys, in a transparent and mobility-enabling fashion for clients. We propose a novel mechanism for persistent, shared key generation and management on the client-side. We then leverage this approach to obtain basic security primitives well suited for securing low-cost transactions which repeatedly execute between a client and a server. Such transactions may span a variety of applications, from two party tasks to elaborated micro-payment schemes involving banks, arbiters, vendor, clients, and more. In particular, concrete applications may include (1) delivery of personalized information by a vendor (via web-pages) which ensures privacy, authenticity, and integrity for each client (e.g., authentication of personalized stock quotes which a vendor sends to a client on a daily/hourly basis); (2) support for secure subscription of such services; (3) delivery of receipts to a client which ensures authenticity and integrity, provable to a third party; and (4) support and integration for (micro)payments, such as SET [SET] and PayWord [RS96].

Our approach has the following noteworthy technical aspects:

1. *Client-side shared key computation:* We propose to use a client-proxy on the client side which transparently computes *modularly secure* shared keys on the client's behalf using the so-called *Janus function* (see Section 2). This computation is based on the server identity, the client identity, and a single secret provided by the client.
2. *Client-side shared key management:* We allow shared keys to persist between browsing sessions of a client. However, a client need not store any shared keys, or any other information. Rather, the (persistent) shared key is recomputed by the client-proxy transparently on demand.
3. *Server-side shared key management:* The server accepts and stores a client's shared key on their first interaction. This is easily integrated into client's records that are typically stored at a server (such records often include usernames, preferences, etc).
4. *Modular structure:* Modularity allows us to adjust the complexity and cost of securing a transaction to the importance and monetary value of the transaction.

The above properties imply that a client need not rely on data stored in memory, and is readily suitable for mobility. The client-proxy that operates on behalf of the client does not maintain any information about the client. Therefore, the client can use various instances of the client-proxy interchangeably. A client can have a copy of the client-proxy on her PC at the office and another copy on her laptop. She can then transparently continue interacting with a server when switching from her PC to her laptop. When on the road, the client may be able to use a client-proxy implemented on an Internet-kiosk placed at the airport, and later use one implemented at an Internet station placed in the hotel.

The client interface is simple. Upon first interaction with the client-proxy (e.g., when starting to run a browser), she provides her identity (e.g., e-mail address) and a secret; she can then reconnect transparently to any server with appropriate session information. Our scheme does not require a client to obtain and maintain her own public and private keys (certificates). The client information (identity, secret) can alternatively be stored on a smart-card, or in a secure file, and be submitted to the client-proxy automatically on the client's behalf. The proposed scheme further gains computational efficiency via minimizing the use of public-key cryptography.

Organization of the Paper: Section 2 presents the client-side generation and management of secret shared keys. In Section 3 we introduce our key establishments and data delivery protocols based on the shared key of Section 2. Section 4 extends our protocols to avoid exposure of the shared key. In Section 5 we show how receipts can be generated to help in potential conflicts between clients and vendors. Finally, we discuss implementation issues in Section 6.

2 Client-side key generation and management

In our scheme, it is up to the client to compute a different secure persistent shared key for each vendor (server) she interacts with. She submits to the vendor an "identity" (e.g., email address) and a shared key, which are to be used by both parties during their subsequent interactions. The shared key is private and should be protected during communication; hence, before submitting the shared key, the client uses the vendor's public key to encrypt it. Public-key encryption is used only during the first interaction with a vendor. After this first exchange, both the client and the vendor can use the secret shared key in their subsequent interactions to authenticate and encrypt data with low computational cost.

An important aspect of this scheme is the method by which a client computes her shared keys. A shared key is computed as a function of three arguments: the client's unique identity (e.g., e-mail address), a (single) secret provided by the client, and the identity of the vendor (expressed, e.g., as as the domain name in the vendor's URL). The function computes a string with the following properties (with respect to an adversary with polynomially bounded computational power):

1. *Secrecy:* An adversary cannot do better than guessing the resulting shared key with negligible probability.
2. *Consistency:* the computed shared key for a given vendor is consistent.
3. *Efficiency:* the computation of the shared key is efficient.
4. *Modular security:* Knowing some of a client's shared keys cannot help an adversary in guessing the client's shared key for a different vendor.
5. *Impersonation resistance:* given a vendor and a client, the adversary cannot do better than guessing another client identity and a corresponding secret, such that the resulting shared keys are identical.

We propose that the shared key computation on the client's behalf is done transparently by a client-proxy. The proxy may be located on the client's machine. Alternatively, it can be located on a different machine with which the client has a trusted communication (e.g., a server within an intranet). Upon first interaction of the client with the client-proxy (e.g., when starting a browser) the client provides a single secret, which the proxy uses thereafter to compute a shared key for each vendor the client interacts with during that browsing session.
For the rest of the paper, we consider the *client* to be the combination of user, the user-interface (e.g., browser), possibly a user-assisting program (e.g., plug-ins to the browser), and the client-proxy. Whenever we say that a client computes or executes an operation, we mean that the computation or execution is done by the client-proxy. Whenever we say the client supplies input (e.g., id or secret), we mean that the user provides the input through the user interface, or that a user-assisting program does it on the user's behalf.

Design of the key generating function To meet the desired properties listed above, we propose to use the *Janus function* $\mathcal{J}$, as defined in [BGGMM97] in the context of personalized interaction. The design of the function $\mathcal{J}$ is based on pseudo-random functions and collision-resistant hash functions (see [GGM86] and [MOV97], respectively). Let h be a collision-resistant hash-function and let f_k be a pseudo-random function chosen from a pseudo-random function ensemble F_l by using k as a seed. Let $||$ denote concatenation and $\otimes$ denote exclusive or. Let id_C denote the identity of the client and let id_V denote the identity of the vendor. Finally, let s_C denote the secret of the client, for which we assume for simplicity $s_C = (s_C^1||s_C^2)$. The Janus function $\mathcal{J}$ is defined as:

$$\boxed{\begin{aligned}&\mathcal{J}(id_C, id_V, s_C) = \\ &\quad h(f_{s_C^1}(id_V))||(f_{s_C^2}(f_{s_C^1}(id_V)) \otimes id_C)\end{aligned}}$$

In [BGGMM97] it is shown that the function $\mathcal{J}$ as defined above satisfies the desired properties for a client password (weak authentication). The quality of a good (machine-generated, non-mnemonic) password and a secret shared key as required here are essentially the same. The length of a shared key is typically in the range of $56 - 128$ bits, which coincides with the output length of a typical hash-function.
A vendor stores each client's identity and the shared key (possibly along with some other data, such as a client's preferences) on the very first interaction with a client, so that it can retrieve the corresponding key upon being presented with a client's identity on a repeat visit.

3 Basic protocols

In this section, we describe the basic protocols used for establishing persistent shared keys, and for subsequent interaction between a client and a server. We also present a model and correctness arguments for our protocols. At the same time, we caution that a careful development of model and correctness proofs (as shown in [BR93] for a simpler, well-known protocol) is beyond the scope of this paper.
First, we present the *Simple Key Establishment Protocol (SKEP)* for establishing relationship between a client and a server, by having the client provide the server with the persistent shared key and some other identifying or payment related information. Then we present the *Simple Data Delivery Protocol (SDDP)* for the subsequent interactions involving data delivery, and the more robust *Extended Data Delivery Protocol (EDDP)*.
In the following, let $E_K(x)$ denote the encryption of a plaintext x with a public-key K, and let $S_k(x)$ denote the signature of x with a private key k. We assume that a client can obtain

each vendor's certified public key, motivated by the emerging public-key infrastructure (see, e.g., "VeriSign.com"). Let $Enc_K(x)$ be a symmetric encryption of x with the shared key K, let MAC_K be a message authentication scheme with a shared key K. Consider two parties, Alice and Bob, that have a shared secret key $K = \langle K_1, K_2 \rangle$. Let $EMAC_K(x) = (Enc_{K_1}(x) || MAC_{K_2}(Enc_{K_1}(x)))$, which can be used in a basic secure communication step between Alice and Bob, that enables delivery of an encrypted, authenticated message x.

3.1 Model

In order to interact, a vendor V and a client C form a "session" s, during which a single shared key is first established and then used. Let the two threads $\Pi^s_{C,V}$ and $\Pi^s_{V,C}$ be the entities involved in session s on the client and, resp., vendor side.
We assume the presence of a polynomially bounded adversary E, which is in charge of the communication (e.g., sending, deleting, reordering of messages) and can execute the following actions:

- *get-private-key*(x): if $x = V$, then E obtains SK_V. If $x = C$, then E obtains s_C
- *get-shared-key*(s): E obtains K of s.
- *compute-*$EMAC_K(x)$: E gets the result of computing $EMAC_K(x)$.

Definition 1 *A vendor (client) x is* **corrupted**, *if a successful* get-private-key(x) *was executed; a session s is* **opened**, *if either a successful* get-shared-key(s) *was executed or either participant is corrupted.*

3.2 Simple Key Establishment Protocol (SKEP)

The SKEP protocol (illustrated in Fig. 1) is used when a client C requests to register (or subscribe) at a vendor V for the first time in order to subscribe. First, the client computes the appropriate persistent shared key $K = \langle K_1, K_2 \rangle$ as $K = \mathcal{J}(id_C, s_C, id_V)$. The component K_1 will be used for encryption, and a component K_2 will be used for authentication. The subsequent message of C to the vendor V contains the persistent shared key K, encrypted via the the vendor public key PK_V, and a random nonce R_C: $(E_{PK_V}(K) || R_C)$. The vendor V then decrypts the first part of the message to obtain K. V replies with $EMAC_K(R_C || R_V)$, where R_V is its own random nonce. C decrypts the message, verifies the MAC and its own random nonce; it then sends the message $EMAC_K(R_V || id_C || I_C)$, where I_C contains possible subscription data, such as start-date or expiration date, and possible payment information, such as credit-card data, SET [SET] payment, data or "commitments" used in electronic (micro-)payments (e.g., as in PayWord [RS96]). V decrypts the message, verifies the MAC, and compares R_V to what it sent earlier; it stores the data id_C and I_C in C's record.

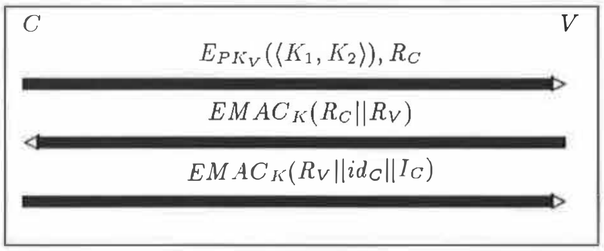

C
V
$E_{PK_V}(\langle K_1, K_2 \rangle), R_C$
$EMAC_K(R_C || R_V)$
$EMAC_K(R_V || id_C || I_C)$

Figure 1: Simple Key Establishment Protocol (SKEP)

Desirable properties of a key establishment protocol include *Key Authentication*, *Entity Authentication*, and *Key Confirmation*, which we now define in turn:

Definition 2 Key Authentication: *For an unopened session s, E can only obtain non-negligible information on K of s.*
Matching Conversation: *A sequence of messages exchanged among $\Pi^s_{C,V}$ and $\Pi^s_{C,V}$, such that each message received by $\Pi^s_{C,V}$ corresponds to the last message sent by $\Pi^s_{V,C}$ and vice versa.*
Entity Authentication: *For an unopened session s, $\Pi^s_{C,V}$ and $\Pi^s_{C,V}$ accept the outcome of the protocol without s having a matching conversation only with negligible probability.*
Key Confirmation: *For an unopened session s, $\Pi^s_{C,V}$ and $\Pi^s_{C,V}$ accept the outcome of the protocol without K being known to other side only with negligible probability.*

We note that the above definition of Entity Authentication is essentially borrowed from [BR93], where a more in-depth discussion and model can be found.

Lemma 3 *SKEP provides Key Authentication, Entity Authentication, and Key Confirmation.*

Proof: For an unopened session s, adversary E can only obtain $E_{PK_V}(K)$. Assuming that the public-key encryption system employed by SKEP is sound, this implies Key Authentication. C

accepts the message of V, only if she can verify $MAC_{K2}(R_C \ldots)$. Given that SKEP assures Key Authentication and that R_C is a random value of "sufficient" length, E can neither compute $MAC_{K2}(R_C \ldots)$ nor guess R_C ahead of time (except with negligible probability) and execute *compute-EMAC*. Similarly, V accepts the message of C, only if he can verify $MAC_{K2}(R_V \ldots)$. Entity Authentication follows. If C successfully verifies $MAC_{K2}(R_C \ldots)$ and decrypts $Enc_{K_1}(R_C \ldots)$, and given Entity Authentication, Key Confirmation follows. □

Note that we define entity authentication, in the sense that a client can be assured that it consistently interacts with vendor, and vice versa. In the SKEP protocol, *identity* information is obtained via a client using a vendor's certified public key and a vendor using a client's payment data, such as credit card or SET data. This is consistent with today's business model of popular web-sites. For instance, on-line subscriptions to the *Wall Street Journal* and to *ESPN Sportzone*, and on-line book purchases at *amazon.com* require only this "weak authentication" from their clients.

3.2.1 Non-repudiation Key Exchange Protocol

SKEP is lacking the *non-repudiation* property, i.e., the possibility for a client to obtain *a receipt*, provable to a third party, for its subscription. SKEP can be extended by one more message exchange to obtain non-repudiation by having the vendor V sign the subscription data of the client; see Fig. 2.

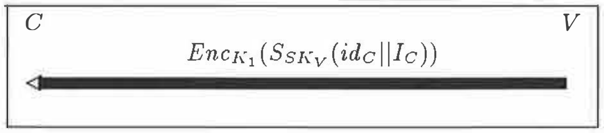


Figure 2: Extension for SKEP

3.3 Simple Data Delivery Protocol (SDDP)

The SDDP protocol (illustrated in Fig. 3) is used when a client C requests from a vendor V some information $D_{C,V}$ (e.g., a personalized web-page). The client sends the request, $R(D_{C,V})$, along with id_C and R_C, where R_C is a random nonce, where the request and the nonce are encrypted and MAC'd. If V finds id_C's key K and I_C assures validity then V replies with $EMAC_K(R_C||D_{C,V})$. The client then decrypts the message, checks that R_C is unchanged, and verifies the MAC.

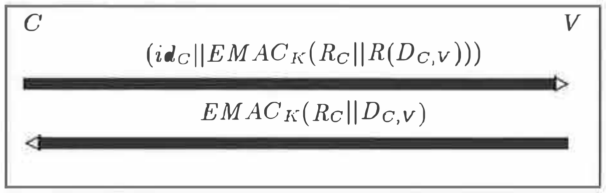


Figure 3: Simple Data Delivery Protocol (SDDP)

Desirable properties of a data delivery protocol include *Data Privacy*, *Entity Authentication*, and *Data Integrity*, which we now define in turn:

Definition 4 Data Privacy*: For an unopened session s, E cannot obtain non-negligible information on* $D_{C,V}$.
Entity Authentication*: see Definition 2.*
Data Integrity*: For an unopened session s, E cannot modify* $D_{C,V}$*, undetectable to C with a non-negligible probability. This also implies that C is assured that the received data is indeed the answer to her request.*

We mention without proof that SDDP fulfills Data Privacy, Entity Authentication for the client and Data Integrity. (The proofs can be derived from the proofs we give in the subsequent section for the EDDP protocol.) However, SDDP provides no Entity Authentication to the vendor. As a consequence E can prompt the vendor via impersonation attacks to send data (even though it might not be readable by E). This might be a problem in terms of chosen message and denial of service attacks. The variant in the next section is more robust in that sense.

3.4 Extended Data Delivery Protocol (EDDP)

The EDDP protocol (illustrated in Fig. 4) requires the client to demonstrate her possession of the appropriate shared key. A request of a client C from a vendor V for information $D_{C,V}$ (e.g., accessing a personalized web-page) is implemented as follows. The client first sends her identity id_C. If V accepts id_C as a client and I_C in the stored record assures validity then V replies with R_V, a random nonce. The client now issues her specific request by replying with $EMAC_K(R_C||R_V||R(D_{C,V}))$, where R_C is the client's random nonce. V checks that R_V is unchanged and verifies the MAC; it replies with

$EMAC_K(R_C||D_{C,V})$. The client checks that R_C is unchanged and verifies the MAC.

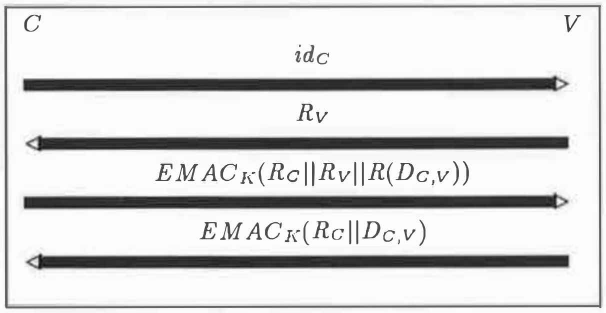


Figure 4: Extended Data Delivery Protocol (EDDP)

Lemma 5 *EDDP provides Data Privacy, mutual Entity Authentication, and Data Integrity.*

Proof: For an unopened session s, E can only obtain the corresponding encryption $Enc_{K_1}(.)$ of either $R(D_{C,V})$ or $D_{C,V}$. Furthermore, these values are prefixed with a random value (R_C), which (for suitable encryption algorithms) makes E executing *compute-EMAC* in advance to verify guesses (especially on $R(D_{C,V})$, a value in a small range) useful only with negligible probability. Thus, Data Privacy follows. Mutual Entity Authentication follows by similar arguments as used in proof of Lemma 3. If C successfully verifies $MAC_{K_2}(R_C||D)$, the Data Integrity is implied, since E cannot compute this MAC, nor guess R_C in advance to make use of *compute-EMAC* with a non-negligible probability. The same argument holds for V. □
EDDP provides *Entity Authentication* to the vendor, before he sends any data and hence gives better protection against chosen plaintext and denial of service attacks. Furthermore, the exposure of the shared key is better protected, since only the corresponding client can prompt the vendor to use it for encryption/authentication. We note that the state of art in modeling and proof techniques (e.g., see [BR93]) does not consider key exposure.

3.5 Discussion

The different applicability of the given protocol-variants can be illustrated by the following examples:
Example 1: Free Personalized Stock Quotes.
id_C is the client's email address and I_C is empty. V is a web-site which provides stock quotes. $D_{C,V}$ is a personalized web-page containing a set of stock quotes for client C. SKEP and SDDP are sufficient.
Example 2: Subscription to Personalized Stock Quotes.
id_C in Step 1 is the client's email address and I_C contains her credit card data and the duration of the subscription. Non-repudiation key exchange is preferable in this case, as it allows a client to use the vendor's signature as a receipt. EDDP is preferable, as the service is restricted to paying users and thus the vendor would like to provide adequate performance.

Computational cost and memory requirement of the protocols: The vendor has to sign a single message per client (none with SKEP). All subsequent communication is done via symmetric encryption/MAC. The client has to encrypt (public-key) a single message per vendor upon first interaction, and only MAC in subsequent interactions with that vendor. In addition, the client has to compute one Janus function per browsing-session and vendor. The client does not need any longterm memory. The vendor needs to store a persistent shared key K for each of its clients. Typically, a vendor stores some information about each of its client in any case, so this does not put an extra burden on the vendor.

4 Avoiding exposure of the persistent shared keys

In this section, we show how to extend the protocols presented in Section 3 so that (1) the exposure of the shared key K is minimized and hence cryptanalysis is made more difficult and (2) reuse of compromised keys is prevented.
We consider the notion of a *session-key* κ, which is used in lieu of the shared key K. The session key is only used for a single instance of SDDP (or EDDP). The session-key is updated by the following zero-message method, originally proposed for SKIP (see [AP95]): $\kappa_n = h(K, n)$, where h is a pseudo-random like function such as MD5 (see [R92]) and n is a strictly monotonically increasing counter. Should a session-key κ_n ever be compromised (for whatever reason) then it cannot be mis-used by an adversary to either decrypt past data transmissions or to forge data in future transmissions.
Note that the above method has the disadvantage of introducing "state" to be kept on both the client and the vendor, namely the counter n. In order to mitigate this drawback, we suggest that the counter be replaced by a strictly monotonic increasing function

of the time (standard GMT), with a pre-specified granularity (e.g., day, hour, or minute) that should depend on the accuracy of the time of the client and server. One complication in this approach is that even with high accuracy, there may be discrepancy between two parties (e.g., around midnight when the unit is a day). To alleviate possible conflicts, we let the client determine the time function and notify the server during a first message. The server will accept the time function received from the client if it is the same as computed locally, or if it is within a (pre-determined) reasonable range from its computed time function (e.g., within an hour).

5 Extending the data delivery protocols to enable receipts

Consider the situation, where a client complains that she paid for a subscription of stock quotes, but never got information from the vendor. A third party cannot decide if indeed the vendor is at fault. We propose to adapt a technique used in the micro-payment protocol "PayWord" (see [RS96]) to obtain the notion of *receipts*.

Assume for example that a subscription is for a month and that it entitles a client C to obtain stock quotes once a day. C chooses at random a value r and sets $w_{31} = r$ and $w_i = h(w_{i+1})$ for a suitable one-way hash function h and for $0 \leq i \leq 30$. C includes w_0 in I_C. Using non-repudiation SKEP gives the client a signature of V on w_0. C confirms the receipt of data (e.g., successful access of her personalized web-page) by sending back w_1, w_2, etc. V can test $w_{i-1} = h(i)$ and only if successful send data the next time. The ith time C acknowledges the receipt by sending w_i and V checks that $w_{i-1} = h(w_i)$. If at any time V's check is not successful, V will stop the session with C and possibly refund C for the rest of the subscription (via some verifiable payment scheme). This option is clearly not in the interest of a vendor, and thus it is unlikely that a vendor will wrongfully claim that he did not get a correct acknowledgment from the client.

The vendor can present w_i as proof that he delivered at least i data items to client C. Client C can present w_0 signed by the vendor as proof that a vendor agreed on a particular chain of receipts. Now if a client rightfully claims that she neither got the information she subscribed to nor any refund from the vendor, then the vendor cannot claim (1) that he did deliver the information, since he cannot compute w_i or (2) a different chain of receipts (since the client has his signature on the correct chain) or (3) that he paid a refund. If, on the other hand, a client wrongfully claimed that she was cheated, then either (1) she cannot show the vendor's signature on an altered chain, (2) the vendor can show w_i on the correct chain (3) or he can prove to have paid a refund.

6 Implementation

Almost all cryptographic methods used in our proposed framework are available in standard cryptographic libraries. The only exception is the function $\mathcal{J}$, which has been implemented within a web-proxy, as part of the Lucent Personalized Web Assistant (LPWA, see [GGMM97]). LPWA is being used by the general public since June 1997 and has been commended on its performance.

The LPWA software also forms the starting point for our first (and currently ongoing) implementation of the protocols presented in this paper. The protocols on the client-side are realized by a web-proxy, just as in LPWA. The client identifies herself to the proxy via proxy-authenticate. The web-proxy implements the Janus-function and embeds the client-side protocols into the client's HTTP-requests before forwarding them to the server. In line with our lightweight approach, HTTP headers are authenticated, but never encrypted. Care is taken that the sever can easily retrieve (and verify) the client's identity. The protocols on the server-side are realized via CGI (and FastCGI, see `http://www.fastcgi.com/`) scripts written in C. The goal of our first implementation is to obtain performance figures on the server-side to shed light on at least the following two issues: (1) Absolute measures of the cryptographic overhead and (2) the cryptographic overhead relative to minimal CGI-scripts (e.g., "Hello World" script) and scripts used in actual servers.

Another interesting direction is to explore ways to integrate our client-side key management scheme with SSL. This potentially yields a scheme which works works with current Web servers, but allows client mobility and requires no secure client-side long-term memory.

Acknowledgments

We thank Daniel Bleichenbacher for helpful discussions on secure protocol design and David M. Kristol and Eran Gabber for advice on software design.

References

[AP95] A. Aziz and M. Patterson, *Design and Implementation of SKIP (Simple Key Management for Internet Protocols)*, In *INET'95* conference.

[BGGMM97] D. Bleichenbacher, E. Gabber, P. Gibbons, Y. Matias, A. Mayer, *A Client-side Cryptographic Engine for Secure Relationships with Multiple Servers*, Bell Labs Technical Memorandum 1997, available at URL `www.bell-labs.com/projects/lpwa/papers.html`.

[BR93] M. Bellare, P. Rogaway, *Entity Authentication and Key Distribution*, Crypto'93 Proceedings, Springer-Verlag.

[GGM86] O. Goldreich, S. Goldwasser, S. Micali, *How to construct random functions*, J. of the ACM, **33**(4), 1986, pp 210 - 217.

[GGMM97] E. Gabber, P. Gibbons, Y. Matias, A. Mayer, *How to Make Personalized Web Browsing Simple, Secure, and Anonymous*, Proc. of Financial Cryptography'97, Springer-Verlag, LNCS 1318. Also available at URL `www.bell-labs.com/projects/lpwa/papers.html`.

[MOV97] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[R92] R. Rivest, *The MD5 Message Digest Algorithm*, Internet-RFC 1321, April 1992.

[R97] R. Rivest, *Perspectives on Financial Cryptography (Invited Lecture)*, Proc. of Financial Cryptography'97, Springer-Verlag, LNCS 1318.

[RS96] R. Rivest and A. Shamir, *PayWord and MicroMint: Two simple micropayment schemes*, 4th Cambridge Workshop on Security Protocols, 1996.

[SET] SET: SECURE ELECTRONIC TRANSACTION Specification.
See, e.g., at URL http://www.visa.com/cgi-bin/vee/sf/set/intro.html?2+0.

[SHTTP95] E. Rescorla and A. Schiffman *The Secure HyperText Transfer Protocol*, Internet-Draft (draft-ietf-wts-shttp-00.txt), July 1995.

[SSH96] T. Ylonen, *SSH - Secure Login Connections over the Internet*, USENIX Workshop on Security, 1996

[SSL96] P. Karlton, A. Freier, and P. Kocher, *The SSL Protocol, 3.0*, Internet Draft, March 1996.

Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers

JavaSoft, Sun Microsystems, Inc.
{gong,mrm,hemma,schemers}@eng.sun.com

Abstract

This paper describes the new security architecture that has been implemented as part of JDK1.2, the forthcoming Java™ Development Kit. In going beyond the sandbox security model in the original release of Java, JDK1.2 provides fine-grained access control via an easily configurable security policy. Moreover, JDK1.2 introduces the concept of protection domain and a few related security primitives that help to make the underlying protection mechanism more robust.

1 Introduction

Since the inception of Java [8, 11], there has been strong and growing interest around the security of Java as well as new security issues raised by the deployment of Java. From a technology provider's point of view, Java security includes two aspects [6]:

- Provide Java (primarily through JDK) as a secure, ready-built platform on which to run Java enabled applications in a secure fashion.
- Provide security tools and services implemented in Java that enable a wider range of security-sensitive applications, for example, in the enterprise world.

This paper focuses on issues related to the first aspect, where the customers for such technologies include vendors that bundle or embed Java in their products (such as browsers and operating systems).

It is worth emphasizing that this work by itself does not claim to break significant new ground in terms of the theory of computer security. Instead, it offers a real world example where well-known security principles [5, 12, 13, 16] are put into engineering practice to construct a practical and widely deployed secure system.

1.1 The Original Security Model

The original security model provided by Java is known as the sandbox model, which exists in order to provide a very restricted environment in which to run untrusted code (called applet) obtained from the open network. The essence of the sandbox model, as illustrated by Figure 1, is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code is not trusted and can access only the limited resources provided inside the sandbox.

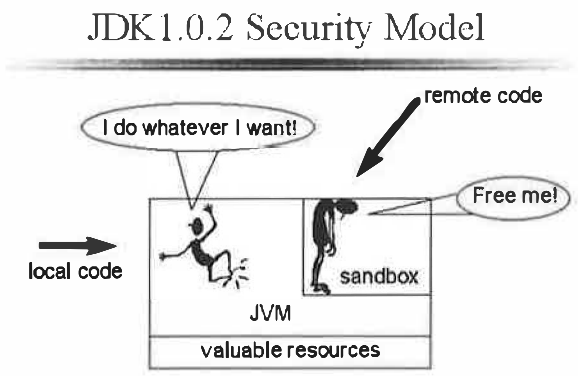


Figure 1: JDK1.0.x Security Model

This sandbox model is deployed through the Java Development Toolkit in versions 1.0.x, and is generally adopted by applications built with JDK, including Java-enabled web browsers.

Overall security is enforced through a number of mechanisms. First of all, the language is designed to be type-safe, and easy to use. The hope is that the burden on the programmer is such that it is less likely to make subtle mistakes, compared with using other programming languages such as C or C++. Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safer code.

Second, compilers and a bytecode verifier ensure that only legitimate Java code is executed. The bytecode verifier, together with the Java virtual machine, guarantees language type safety at run time.

Moreover, a class loader defines a local name space, which is used to ensure that an untrusted applet cannot interfere with the running of other Java programs.

Finally, access to crucial system resources is mediated by the Java virtual machine and is checked in advance by a **SecurityManager** class that restricts to the minimum the actions of untrusted code.

JDK1.1.x introduced the concept of signed applet. In this extended model, as shown in Figure 2, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format.

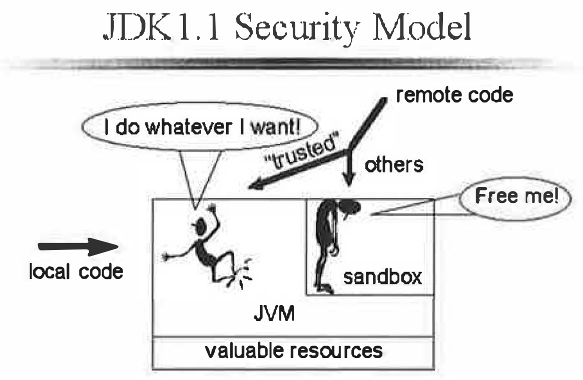


Figure 2: JDK1.1 Security Model

The rest of this paper focuses on the new system security features. Discussion of various language safety issues can be found elsewhere (e.g., [3, 4, 19, 21]).

1.2 Evolving the Sandbox Model

The new security architecture in JDK1.2, as illustrated in Figure 3, is introduced primarily for the following purposes.

- Fine-grained access control.

  This capability has existed in Java from the beginning, but to use it, the application writer has to do substantial programming (e.g., by subclassing and customizing the **SecurityManager** and **ClassLoader** classes).

  HotJava is such an example application. However, such programming is extremely security sensitive and requires sophisticated skills and in-depth knowledge of computer security. The new architecture makes this exercise simpler and safer.

- Easily configurable security policy.

  Once again, this feature exists in Java but is not easy to use. This design goal implies that the security and its implementation or enforcement mechanism should be clearly separated. Moreover, because writing security code is not straightforward, it is desirable to allow application builders and users to configure security policies without having to program.

- Easily extensible access control structure.

  Up to JDK1.1, to create a new access permission, one has to add a new **check()** method to the **SecurityManager** class. The new architecture allows typed permissions and automatic handling. No new method in the **SecurityManager** class needs to be created in most cases. (Actually, we have not encountered a situation where a new method must be created.)

- Extension of security checks to all Java programs, including applets as well as applications.

  There should not be a built-in concept that all local code is trusted. Instead, local code should be subjected to the same security controls as applets, although one should have the choice

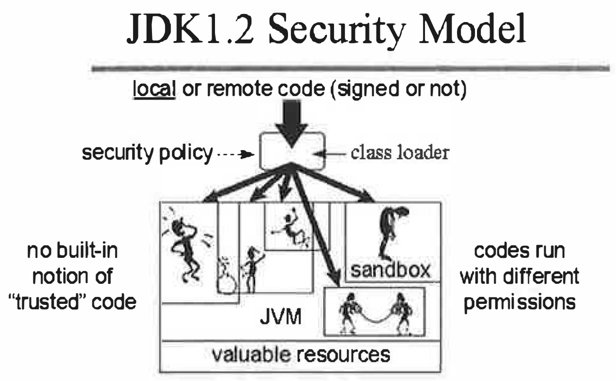

JDK1.2 Security Model
local or remote code (signed or not)
security policy
class loader
no built-in notion of "trusted" code
sandbox
JVM
codes run with different permissions
valuable resources

Figure 3: JDK1.2 Security Model

to declare that the policy on local code (or remote code) be the most liberal (thus local code effectively runs as totally trusted). The same principle applies to signed applets and applications.

Finally, we also take this opportunity to make internal structural adjustment in order to reduce the risks of creating subtle security holes in programs. This effort involves revising the design and implementation of the `SecurityManager` and `ClassLoader` classes as well as the underlying access control checking mechanism.

1.3 Related Work

The fundamental ideas adopted in the new security architecture have roots in the last 40 years of computer security research, such as the overall idea of access control list [10]. We followed some of the Unix conventions in specifying access permissions to the file system and other system resources, but significantly, our design has been inspired by the concept of protection domains and the work dealing with mutually suspicious programs in Multics [17, 15], and right amplification in Hydra [9, 20].

One novel feature, which is not present in operating systems such as Unix or MS-DOS, is that we implement the least-privilege principle by *automatically* intersecting the sets of permissions granted to protection domains that are involved in a call sequence. This way, a programming error in system or application software is less likely to be exploitable as a security hole.

Note that although the Java Virtual Machine (JVM) typically runs over another hosting operating system such as Solaris, it may also run directly over hardware as in the case of the network computer JavaStation running JavaOS [14]. To maintain platform independence, our architecture does not depend on security features provided by an underlying operating system.

Furthermore, our architecture does not override the protection mechanisms in the underlying operating system. For example, by configuring a fine-grained access control policy, a user may grant specific permissions to certain software, but this is effective only if the underlying operating system itself has granted the user those permissions.

Another significant character of JDK is that its protection mechanisms are language-based, within a single address space. This feature is a major distinction from more traditional operating systems, but is very much related to recent works on software-based protection and safe kernel extensions (e.g., [2, 1, 18]), where various research teams have lately aimed for some of the same goals with different programming techniques.

2 New Protection Mechanisms

This section covers the concept and implementation of some important new primitives introduced in JDK1.2, namely, security policy, access permission, protection domain, access control checking, privileged operation, and Java class loading and resolution.

2.1 Security Policy

There is a system security policy, set by the user or by a system administrator, that is represented by a policy object, which is instantiated from the class `java.security.Policy`. There could be multiple instances of the policy object, although only one is "in effect" at any time. This policy object maintains a runtime representation of the policy, is typically instantiated at the Java virtual machine start-up time, and can be changed later via a secure mechanism.

In abstract terms, the security policy is a mapping from a set of properties that characterize running code to a set of access permissions that is granted to the concerned code.[1]

Currently, a piece of code is fully characterized by its origin (its location as specified by a URL) and the set of public keys that correspond to the set of private keys that have been used to sign the code using one or more digital signature algorithms. Such characteristics are captured in the class `java.security.CodeSource`, which can be viewed as a natural extension of the concept of a code base within HTML. (It is important not to confuse `CodeSource` with the `CodeBase` tag in HTML.) Wild cards are used to denote "any location" or "unsigned".

Informally speaking, for a code source to match an entry given in the policy, both the URL information and the signature information must match. For URL matching, if the code source's URL is a prefix of an entry's URL, we consider this a match. For signature matching, if one public key corresponding to a signature in the code source matches the key of a signer in the policy entry, we consider it a match.

[1]In the future, the security policy can be extended to include and consider information such user authentication and delegation.

When a code source matches multiple policy entries, for example, when the code is signed with multiple signatures, permissions granted are additive in that the code is given all permissions contained in all the matching entries. For example, if code signed with key A gets permission X and code signed by key B gets permission Y, then code signed by both A and B gets permissions X and Y.

Verification of signed code uses a new package of certificate `java.security.cert` that fully supports the processing of X.509v3 certificates.

The policy within the Java runtime is set via a programming API. We also specify an external policy representation in the form of an ASCII policy configuration file. Such a file essentially contains a list of entries, each being a pair, consisting of a code source and its permissions. In such a file, a public key is signified by an alias – the string name of the signer – where we provide a separate mechanism to create aliases and import their matching public keys and certificates.

2.2 Permission

We have introduced a new hierarchy of typed and parameterized access permissions that is rooted by an abstract class `java.security.Permission`. Other permissions are subclassed either from the `Permission` class or one of its subclasses, and generally should belong to their own packages.

For example, the permission representing file system access is located in the Java I/O package, as `java.io.FilePermission`. Other permission classes that are introduced in JDK1.2 include: `java.net.SocketPermission` for access to network resources, `java.lang.RuntimePermission` for access to runtime system resources such as properties, and `java.awt.AWTPermision` for access to windowing resources. In other words, access methods and parameters to most of the controlled resources, including access to Java properties and packages, are represented by the new permission classes.

A crucial abstract method in the `Permission` class that needs to be implemented for each new class of permission is the `implies` method. Basically, `a.implies(b) == true` means that, if one is granted permission `a`, then one is naturally granted permission `b`. This is the basis for all access control decisions.

For convenience, we also created abstract classes `java.security.PermissionCollection` and `java.security.Permissions` that are subclasses of the `Permission` class. `PermissionCollection` is a collection (i.e., a set that allows duplicates) of `Permission` objects for a category (such as `FilePermission`), for ease of grouping. `Permissions` is a heterogeneous collection of collections of `Permission` objects.

Not every permission class must support a corresponding collection class. When they do, it is crucial to implement the correct semantics for the `implies` method in the corresponding permission collection classes. For example, FilePermission can get added to the `FilePermissionCollection` object in any order, so the latter must know how to correctly compare a permission with a permission collection.

Typically, each permission consists of a target and an action thus, informally, a permission implies another if and only if both the target and the action of the former respectively implies those of the latter.

Take `FilePermission` for example. There are two kinds of targets: a directory and a file. There are four ways to express a file target: `path`, `path/file`, `path/*`, and `path/-`. `path/*` denotes all files and directories in the directory `path`, and `path/-` denotes all files and directories under the subtree of the file system starting at `path`. The actions include `read`, `write`, `execute`, and `delete`.

Therefore, "read file /tmp/abc" is a permission, and can be created using the following Java code:

```
p = new FilePermission("/tmp/abc", "read");
```

Permission (`/tmp/*`, `read`) implies permission (`/tmp/abc`, `read`), but not vice versa. Permission (`/home/gong/-`, `read,write`) implies permission (`/home/gong/public_html/index.html`, `read`).

In the case of `SocketPermission`, a net target consists of an IP address and a range of port numbers. Actions include `connect`, `listen`, `accept`, and others. One SocketPermission implies another if and only if the former covers the same IP address and the port numbers for the same set of actions.

Applications are free to add new categories of permissions. Note that a piece of Java code can create any number of permission objects, but such actions do not grant the code the corresponding access rights. What matters is that permission objects the Java runtime system associates with the Java code through the concept of protection domains.

2.3 Protection Domain

A new class `java.security.ProtectionDomain` is package-private, and is transparent to most Java developers. It serves as a useful level of indirection in that permissions are granted to protection domains, to which classes and objects belong, and not to classes and objects directly.[2] In other words, a domain can be scoped by the set of objects that correspond to a principal, where a principal is an entity in the computer system to which authorizations (and as a result, accountability) are granted [16]. The Java sandbox in JDK1.0.2 is one example of a protection domain with a fixed boundary.

In JDK1.2, protection domains are created "on demand", based on code source. Each class belongs to one and only one domain. The Java runtime maintains the mapping from code (classes and objects) to their protection domains and then to their permissions.

Protection domain also serves as a convenient point for grouping and isolation between units of protection within the Java runtime. For example, it is possible to separate different domains from interacting with each other. Any permitted interaction must be either through system code or explicitly allowed by the domains concerned.

The above point brings up the issue of accessibility, which is orthogonal to security. In the Java virtual machine, a class is distinguished by itself plus the class loader instance that loaded the class. In other words, a class loader defines a distinct name space and can be used to isolate and protect code within one protection domain if the loader refuses to load code from different domains (and with different permissions).

On the other hand, it is sometimes desirable to allow code from different domains to interact with each other – for example, in the case of an application made up from Java Beans signed by different public keys, the beans should be able to access each other (which is the purpose of the application) although the runtime environment may insist that different beans are loaded into different domains. The **`AppletClassLoader`** class used by the **`appletviewer`** in JDK1.2 will load classes from different domains.

One protection domain is special: the system domain, which consists of system code that is loaded with a **`null`** class loader (basically all classes located on **`CLASSPATH`**) and is given special privileges. It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, are directly accessible only via system code.

[2] In the future, protection domains can be further characterized by user authentication and delegation so that the same code could obtain different permissions when running "on behalf of" of different principals.

2.4 Domain-Based Access Control

The decision of granting access to controlled resources can only be made within the right context, which must provide answers to questions such as "who is requesting what, on whose behalf". Often, a thread is the right context for access control. Less frequently, access control decisions have to be carried out among multiple threads that must cooperate in obtaining the right context information. This section focuses on the former, as it is the most common case encountered in building JDK1.2.

A thread of execution may occur completely within a single protection domain (i.e., all classes and objects involved in the thread belong to the identical protection domain) or may involve multiple domains such as an application domain and also the system domain.

For example, an application that prints a message out will have to interact with the system domain that is the only access point to an output stream. In this case, it is crucial that at any time the application domain does not gain additional permissions by calling the system domain. Otherwise, there can be security serious implications.

In the reverse situation where a system domain invokes a method from an application domain, such as when the AWT system code calls an applet's **`paint`** method to display the applet, it is again crucial that at any time the effective access rights are the same as current rights enabled in the application domain.

In other words, a less "powerful" domain cannot gain additional permissions as a result of calling a more powerful domain; whereas a more powerful domain must lose its power when calling a less powerful domain. This principle of least privilege is applied to a thread that transverses multiple protection domains.

Up to JDK1.1, any code that performs an access control decision relies on explicitly knowing its caller's status (i.e., being system code or applet code). This is fragile in that it is often insufficiently secure to know only the caller's status but also the caller's caller's status and so on. At this point, placing this discovery process explicitly on the typical programmer becomes a serious burden, and can be error-prone.

To relieve this burden by automating the access checking process, JDK1.2 introduces a new class **`java.security.AccessController`**. Instead of trying to discover the history of callers and their status within a thread, any code can query the access controller as to whether a permission would succeed if performed right now. This is done by calling the **checkPermission** method of the **AccessController** class with a **Permission** object that represents the permission in question.

By default, the access controller will return silently only if all callers in the thread history (e.g., all classes on the call stack) belong to domains that have been granted the said permission. Otherwise, it throws a **java.security.AccessControlException**, which is a subclass of **java.lang.SecurityException**, usually printing the reason of denial.

This default behavior is obviously the most secure but is limiting in some cases where a piece of code wants to temporarily exercise its own permissions that are not available directly to its callers. For example, an applet may not have direct access to certain system properties, but the system code servicing the applet may need to obtain some properties in order to complete its tasks.

For such exceptional cases, we provide a primitive, via static methods **beginPrivileged** and **endPrivileged** in the **AccessController** class. By calling **beginPrivileged**, a piece of code is telling the Java runtime system to ignore the status of its callers and that it itself is taking responsibility in exercising its permissions.

To summarize, a simple and prudent rule of thumb for calculating permissions is the following:

- The permission of an execution thread is the intersection of the permissions of all protection domains transversed by the execution thread.
- When some code calls the **beginPrivileged** primitive, the permission of the execution thread includes a permission if it is allowed by the said code's protection domain and by all protection domains that are called or entered directly or indirectly subsequently.
- When a new thread is created, it inherits from its parent thread the current security context (i.e., the set of protection domains present in the parent at child creation time). This inheritance is transitive.

In following the above rule, the access controller examines the call history and the permissions granted to the relevant protection domains, and to return silently if the request is granted or throw a security exception if the request is denied.

There are two obvious strategies for implementing this access control rule. In an "eager evaluation" implementation, whenever a thread enters a new protection domain or exits from one, the set of effective permissions is updated dynamically. The benefit is that checking whether a permission is allowed is simplified and can be faster in many cases. The disadvantage is that, because permission checking occurs much less frequently than cross-domain calls, a large percentage of permission updates may be useless effort.

JDK1.2 employs a "lazy evaluation" implementation where, whenever a permission checking is requested, the thread state (as reflected by the current thread stack or its equivalent) is examined and a decision is reached to either deny or grant the particular access requested. One potential downside of this approach is performance penalty at permission checking time, although this penalty would have been incurred anyway in the "eager evaluation" approach (albeit at earlier times and spread out among each cross-domain call). In our implementation, performance of this algorithm is quite acceptable[3], so we feel that lazy evaluation is the most economical approach overall.

2.5 Revised SecurityManager

Up to JDK1.1., when access to a critical system resource (such as file I/O and network I/O) is requested, the resource handling code directly or indirectly invokes the appropriate **check** method on the installed **java.lang.SecurityManager** to evaluate the request and decide if the request should be granted or denied.

JDK1.2 maintains backward compatibility in that all **check()** methods in **SecurityManager** are still supported, but we have changed their default implementations to invoke **AccessController**, whenever feasible, with the appropriate permission object. This class, which has been abstract up to JDK1.1.x, is made concrete in JDK1.2.

To illustrate the usage of the new access control mechanism, let us examine a small example for checking file access. In earlier versions of the JDK, the following code is typical:

```
ClassLoader loader =
        this.getClass().getClassLoader();
if (loader != null) {
    SecurityManager security =
        System.getSecurityManager();
    if (security != null) {
        security.checkRead("path/file");
    }
}
```

[3]For details of the implementation of protection domain, and a discussion on performance and optimization techniques, please refer to [7].

Under the new architecture, the check typically should be invoked whether or not there is a classloader associated with a calling class. It should be simply:

```
FilePermission p =
    new FilePermission("path/file", "read");
AccessController.checkPermission(p);
```

Note that there are legacy cases (for example, in some browsers) where whether there is an instance of the **SecurityManager** class installed signifies one or the other security state that may result in different actions being taken. We currently do not change this aspect of the **SecurityManager** usage, but would encourage application developers to use the techniques introduced in this new version of the JDK in their future programming.

Moreover, we have not revised system code to always call **AccessController** (and not checking for the existence of a classloader), because of the potential of existing software subclassing the **SecurityManager** and customizing these check methods.

To use the privilege primitive, the following code sample should be followed:

```
try {
    AccessController.beginPrivileged();
    (some sensitive code)
} finally {
    AccessController.endPrivileged();
}
```

Some important points about being privileged. Firstly, this concept only exists within a single thread. That is, a protection domain being so privileged is scoped by the thread within which the call to become privileged is made. Other threads are not affected.

Secondly, in this example, the body of code within try-finally is privileged. However, it will lose its privilege if it calls (from within the privileged block) code that is less privileged.

Moreover, although it is a good idea to use **beginPrivileged** and **endPrivileged** in pairs as this clearly scopes the privileged code, we have to deal with the case when **endPrivileged** is not called, because forgetting to disable a privilege can be very dangerous. To reduce or eliminate the risk, we have put in additional mechanism to safe guard this primitive.

2.6 Secure Class Loading

The class **java.security.SecureClassLoader** is a concrete implementation of the abstract class **java.lang.ClassLoader** that loads classes and records the protection domains they belong to. It also provides methods to load a class from bytecode stored in a byte array, an URL, and an **InputStream**. This class can be extended to include new methods, but most existing methods are final, as this class is significant for security.

All applets and applications (except for system classes) are loaded by a SecureClassLoader either directly or indirectly (in which case, it is probably loaded by another classloader that itself is loaded by a SecureClassLoader).

SecureClassLoader's **loadClass** methods enforce the following search algorithm where, if the desired class (by the given name) is not found, the next step is taken. If the class is still not found after the last step, a **ClassNotFoundException** is thrown.

1. See if the class is already loaded and resolved
2. See if the class requested is a system class. if so, load the class with the null system classloader.
3. Attempt to find the class in a customizable way, using a non-final method **findAppClass**, which by default will try to find the class in a second local search path that is defined by a property named **java.app.class.path**.

Note that in step 2, all classes on the search path CLASSPATH are treated as system classes, whereas in step 3, all classes on the search path **java.app.class.path** are considered non-system classes.[4]

Programmers who must write class loaders should, whenever feasible, subclass from the concrete **SecureClassLoader** class, and not directly from the abstract class **java.lang.ClassLoader**.

A subclass of **SecureClassLoader** may choose to overwrite the **findAppClass** method in order to customize class searching and loading. For example, the **AppletClassLoader** caches all raw class materials found inside a JAR file. Thus, it is reasonable for the **AppletClassLoader**, which is a subclass of the **SecureClassLoader**, to use **findAppClass** to look into its own cache. A class introduced in such a fashion is guaranteed not to be a system class, and is subjected to the same security policy as its loading class.

[4]The path **java.app.class.path** is currently specified in a platform dependent format. There might be a future need to develop a generic **Path** class that not only provides platform independent path names but also makes dynamical path manipulation easier.

Often a class may refer to an another class and thus cause the second class belonging to another domain to be loaded. Typically the second class is loaded by the same classloader that loaded the first class, except when either class is a system class, in which case the system class is loaded with a null classloader.

2.7 Extending Security to Applications

To apply the same security policy to applications found on the local file system, we provide a new class `java.security.Main`, which can be used in the following fashion in place of the traditional command `java application` to invoke a local application:

```
java java.security.Main application
```

This usage makes sure that any local application on the `java.app.class.path` is loaded with a `SecureClassLoader` and therefore is subjected to the security policy that is being enforced. Clearly, non-system classes that are stored on the local file system should all be on this path, not on the `CLASSPATH`.

3 Discussion

In this section, we discuss a number of open questions and possible improvement to the current architecture. But we start by discussing how a developer or user is impacted by the new architecture.

3.1 Utilizing the New Architecture

For a user of the built-in `appletviewer` or a new version of a browser that deploys this new security architecture, the user can continue to do things the same way as before, which means that the same policy in JDK1.1.x will apply.

On the other hand, a "power user" can use the `PolicyTool` built-in for JDK1.2 (or an equivalent one shipped with the browser) to customize the security policy, thus utilizing the full benefit of the new security architecture. Such customization may involve setting up a certificate store, which can be done via the `KeyTool`.

The typical application developer, in general, needs to do nothing special because, when the application is run on top of JDK1.2, the security features are invoked automatically. Except that the developer might want to use the built-in tools to package the resulting application into JAR files, and may choose to digitally sign them.

For a software library developer whose code controls certain resources, the developer may need to extend the existing permission class hierarchy to create application-specific permissions. The developer may also need to learn to use features provided by the `AccessController` class, such as the privilege primitive.

3.2 Handling Non-Class Content

When running applets or applications with signed content, the JAR and Manifest specifications on code signing allow a very flexible format. Recall that classes within the same archive can be unsigned, signed with one key, or signed with multiple keys. Other resources within the archive, such as audio clips and graphic images, can also be signed or unsigned.

This flexibility brings about the issue of interpretation. The following questions need to be answered, especially when not all signatures are granted the same privileges. Should images and audio clips be required to be signed with the same key if any class in the archive is signed? If images and audio files are signed with different keys, can they be placed in the same `appletviewer` (or browser page), or should they be sent to different viewers?

These questions are not easy to answer, and require consistency across platforms and products to be most effective. Our intermediate approach is to provide a simple answer – all images and audio clips are forwarded to be processed whether they are signed or not. This temporary solution will be improved once a consensus is reached.

3.3 Enabling Fine-Grained Privileges

The privileged primitive discussed earlier in a sense "enables" all permissions granted to a domain. We can contemplate to enrich the construct so that a protection domain can request to enable privilege for only some of its granted permissions. This should further reduce the security impact of making a programming mistake. For example, the code segment below illustrates how to turn on the privilege of only reading everything in the "/tmp" directory.

```
FilePermission p =
    new FilePermission("/tmp/*", "read");
try {
    AccessController.beginPrivileged(p);
    some sensitive code
} finally {
    AccessController.endPrivileged(p);
}
```

3.4 Extending Protection Domains

The first possibility is to subdivide the system domain. For convenience, we can think of the system domain as a single, big collection of all system code. For better protection, though, system code should be run in multiple system domains, where each domain protects a particular type of resource and is given a special set of rights. For example, if file system code and network system code run in separate domains, where the former has no rights to the networking resources and the latter has no rights to the file system resources, the risks and consequence of an error or security flaw in one system domain is more likely to be confined within its boundary.

Moreover, protection domains currently are created transparently as a result of class loading. It might be desirable to provide explicit primitives to create a new domain. Often, a domain supports inheritance in that a sub-domain automatically inherits the parent domain's security attributes, except in certain cases where the parent further restricts or expands the sub-domain explicitly.

Finally, each domain (system or application) may also implement additional protection of its internal resources within its own domain boundary. Because the semantics of such protection is unlikely to be predictable by the JDK, the protection system at this level is best left to the application developers. Nevertheless, JDK1.2 provides `SignedObject`, `Guard`, and `GuardedObject` classes that simplify a developer's task.

4 Summary and Future Work

This paper gives an overview of the motivation and the new security architecture implemented in JDK1.2. Although we do not break new theoretical ground in computer security, we attempt to distill the best practices from research in the past four decades, such as clear separation between security policy and implementation, and engineer them into a widely deployed programming platform. Our implementation has a number of novel aspects that demonstrate beyond the doubt the efficiency of language-based protection mechanisms. The success of this development effort also highlights the excellent extensibility of the Java platform.

In future releases, we are investigating user authentication techniques, an explicit principal concept, a general mechanism for cross-protection-domain authorization, and the "running-on-behalf" style delegation. We are also working towards additional features such as arbitrary grouping of permissions, the composition of security policies, and resource consumption management, which is relatively easy to implement in some cases, e.g., when limiting the number of windows any application can pop up at any one time, but more difficult in other cases, e.g., when limiting memory or file system usage.

Acknowledgments

Additional members of the JavaSoft security group, including Gigi Ankeny, Charlie Lai, Jan Luehe, and Jeff Nisewanger, made significant contributions during the course of the design and implementation of new security features in JDK1.2. Other members of the JavaSoft community, notably Josh Bloch, Sheng Liang, Roger Riggs, Nakul Saraiya, and Bill Shannon, provided invaluable insight, detailed reviews, and much needed technical assistance.

We are grateful for support from Dick Neiss, Jon Kannegaard, and Alan Baratz, for technical guidance from James Gosling, Graham Hamilton, and Jim Mitchell, and for indispensable collaboration from the testing and documentation groups. We received numerous suggestions from our corporate partners and licensees, whom we could not fully list here.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Colorado, December 1995. Published as ACM Operating System Review 29(5):251–266, 1995.

[2] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems,* 12(4):271–307, November 1994.

[3] D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997.

[4] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1996.

[5] M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold Co., New York, 1988.

[6] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.

[7] L. Gong and R. Schemers. Implementing Protection Domains in the Java™ Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.

[8] J. Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, Menlo Park, California, August 1996.

[9] A.K. Jones. *Protection in Programmed Systems.* Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA 15213, June 1973.

[10] B.W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971. Reprinted in ACM Operating Systems Review, 8(1):18–24, January, 1974.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Menlo Park, California, 1997.

[12] P.G. Neumann. *Computer-Related Risks.* Addison-Wesley, Menlo Park, California, 1995.

[13] U.S. General Accounting Office. Information Security: Computer Attacks at Department of Defense Pose Increasing Risks. Technical Report GAO/AIMD-96-84, Washington, D.C. 20548, May 1996.

[14] S. Ritchie. Systems Programming in Java. *IEEE Micro*, 17(3):30–35, May/June 1997.

[15] J.H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[16] J.H. Saltzer and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[17] M.D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility.* Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1972.

[18] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996. Published as *ACM Operating Systems Review*, 30, special winter issue, 1996.

[19] T. Thorn. Programming Languages for Mobile Code. *ACM Cumpting Surveys*, 29(3):213–239, September 1997.

[20] W.A. Wulf, R. Levin, and S.P. Harbison. *HYDRA/C.mmp – An Experimental Computer System.* McGraw-Hill, 1981.

[21] F. Yellin. Low Level Security in Java. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.

Secure Public Internet Access Handler (SPINACH)

Elliot Poger, Mary G. Baker
Computer Science Department, Stanford University
{elliot,mgbaker}@mosquitonet.stanford.edu

Abstract: *This paper describes a system that controls access to computer networks through publicly accessible LANs, enabling network administrators to authorize users either on a permanent or occasional basis. The system has been designed with minimal assumptions about the software and hardware required of users, and requires very little specialized equipment within the network infrastructure. We enumerate the requirements for such a system, describe the design and implementation of the system, and note tradeoffs between security and efficiency.*

1. Motivation

In early 1996, Stanford University completed a new building to house its Computer Science Department. The new building includes Ethernet ports in every office, as well as in various public spaces: meeting rooms, lobbies, and lounges. Unfortunately, 18 months after the building opened, concerns about unauthorized users tapping into the department network have prevented the activation of network connections in publicly accessible areas ("public ports"). Similar problems plague many other buildings, especially on college campuses, where the desire for mobile connectivity is high but physical security is lax. Even though building designers had the foresight to include network connections in many parts of these buildings, political and security considerations have led to a frustrating waste of potential network connectivity. Those who desire network connectivity in public parts of the building are forced to use wireless network connections, which are often slow and expensive.

There are several reasons Stanford University, and the Computer Science Department in particular, do not want to allow unknown users access to the building network. Most importantly, we do not want to allow rogue users to attack other computers connected to the building network in offices and labs. Although hackers can already attack department computers over the Internet, we do not want to make these attacks, as well as eavesdropping on network traffic, any easier by allowing them access within our network. Also, some network services outside our department use the source IP address of transmissions to grant access. For example, some Internet services have been licensed for use at Stanford University and are made available to any host with a Stanford IP address, and we are obligated to prevent abuse of these licenses. In general, we want to minimize the chances that someone will misuse the Internet from a Stanford IP address, and if this misuse does occur, we want to identify the perpetrator so that we can hold him accountable. Perhaps less of a concern is that of bandwidth—we don't want to allow unauthorized users to degrade everyone else's service in the building by using network bandwidth to which they are not entitled. Since physical security in the building is minimal, as it is in many universities, libraries, and public institutions, we need a mechanism for restricting access through public network ports if these ports are to be activated.

Once we have an access control mechanism in place, we can allow specifically authorized users to connect to the high-bandwidth wired network in the building from public ports without compromising network security. To provide this access control, we have constructed the *Secure Public Internet Access Handler* (SPINACH). In SPINACH, a self-configuring router controls per-user access from a public subnet to a private one, using Kerberos or a similar mechanism to authenticate users and provide an audit path before users are granted access. With the exception of one custom software component on the router, SPINACH uses only standard protocols and software and requires only minimal software (telnet or web clients) on users' machines.

The SPINACH system establishes a "prisonwall," controlling the flow of packets between those hosts connected to public ports and the rest of the building network. As opposed to a firewall, which protects machines *inside* a particular network from malicious users *outside* the network [2][4], the prisonwall protects machines *outside* one portion of a network by refusing to forward packets that come from unauthorized hosts within. As users within the prisonwall authenticate themselves and thus activate network access for their hosts, SPINACH maintains an audit trail so that the users can be held accountable for traffic they generate on the network.

SPINACH has been designed with minimal assumptions regarding the network hardware available as well as the software installed on users' machines, so that it can be installed in a wide variety of institutions and require little ongoing oversight from network administrators. As such, it does not provide as high a level of security as some access control systems; however, it provides a useful level of security without requiring expensive network equipment or custom client software, and thus may be the most appropriate method of access control for some networks.

In this paper, we describe the design and implementation of the SPINACH system. Section 2 outlines the system requirements and policies. In Section 3, we describe the interfaces through which network users and administrators interact with SPINACH. Section 4 discloses the details of how we implement these policies and interfaces. The remainder of the paper describes the security tradeoffs in SPINACH, other systems with aims similar to ours, some possible future improvements to SPINACH, and conclusions we have drawn through this research.

2. System Requirements, Policies, and Definitions

The SPINACH system has two major functions: it controls the passage of network communications between public ports and the rest of the building network, and it provides a mechanism for unknown users to prove themselves as authorized so that they can have full network access. Both functions are implemented on the same network host, the *SPINACH router*. This section describes the requirements that the SPINACH router must fulfill, and the facilities that must be present within the network infrastructure and on hosts connected to public ports in order to implement both functions. SPINACH has been designed to require no special software on computers that users connect to public ports, and to require as little as possible of the network infrastructure, so that it can be deployed in any network installation with minimal expenditure of time and money.

2.1 Network Arrangement

The SPINACH system consists of a collection of public network ports on one or more LANs. These LANs are connected to the surrounding network infrastructure

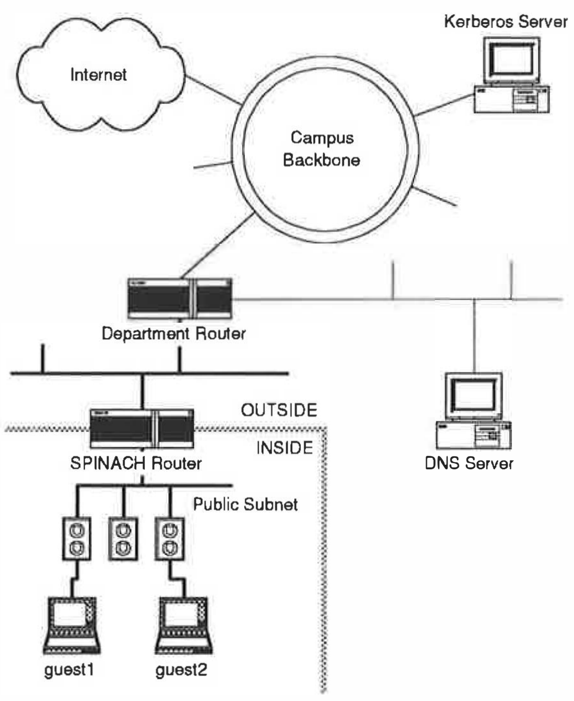


FIGURE 1. Network and security arrangement of the SPINACH system. The gray line running through the SPINACH router illustrates the prisonwall boundary, which separates the public subnet (inside) from the network as a whole (outside).

through a SPINACH router. The SPINACH router, an IP-routing Unix host (fully described in Section 4.1), forwards data packets between hosts on these public LANs and the outside networks. For routing purposes, hosts connected to the public ports are grouped into one or more IP subnets.

In our deployed SPINACH prototype (see Figure 1), the public ports are Ethernet ports located in publicly accessible areas of our building. These Ethernet ports are connected by a VLAN switch, so that data flows between them as if they were on the same LAN segment. Hosts connected to the public ports (labeled as "guest1" and "guest2" in Figure 1) are assigned addresses from one subnet, which we refer to as the "public subnet." The SPINACH router is connected to the same VLAN so it can route packets between the public subnet and the rest of the building network. In other SPINACH installations, some type of LAN other than Ethernet could be used, more than one LAN could be used to connect the public ports, and hosts could be arranged into more than one IP subnet, but for the purposes of this paper we assume the arrangement of our prototype system. Changing these parameters would require slight modifications to the routing and filtering software on the SPINACH router, but the system would function in basically the same way. For example, even a wireless LAN such as WaveLAN could be used for the public subnet, so long as the SPINACH system software were modified to accept WaveLAN, rather than Ethernet, link-layer addresses.

Figure 1 also shows the department Domain Name Service (DNS) server and campus Kerberos server. The Kerberos server provides authentication services for users affiliated with the University. Some other authentication service could work as well, with modifications to the user-authorization software on the SPINACH router; in this paper, we assume the use of Kerberos. The DNS server is needed for hosts on the public subnet to find the IP address of the campus Kerberos server.

Because all packets that travel between hosts on the public network ports ("inside the prisonwall") and hosts elsewhere ("outside the prisonwall") must be forwarded through the SPINACH router, the SPINACH router can filter out all packets that are deemed dangerous. The SPINACH router creates a security boundary between the public Ethernet ports and all other networks.

2.2 Security Policy

Being a research institution, we do not want to squelch the development or use of new network applications by instituting overly specific rules regarding exactly what traffic is allowed on the public subnet [5]. Thus, rather than taking the typical firewall approach by allowing only the use of certain prescribed protocols through proxies running at the security boundary, we filter traffic on a per-*user* basis. We restrict use of the network through public ports to those people whom we can hold accountable for their actions. The SPINACH router allows these trusted users unrestricted access to the network and prevents untrusted users from accessing the network at all.

Traffic to and from hosts within the public subnet can be divided into three types. *Outgoing* traffic travels from within the public subnet to hosts outside. *Incoming* traffic comes from hosts outside the public subnet and is destined for hosts within. *Internal* traffic moves between two hosts on the public subnet. The SPINACH router uses different packet-filtering policies for incoming and outgoing traffic, following a particular set of rules to determine whether a given packet will be forwarded towards its destination or dropped. Internal traffic is not affected by the SPINACH router at all.

The SPINACH router forwards all *outgoing* traffic from those hosts on the public subnet which a user has authorized using the procedure described in Section 3.2. All outgoing packets from unauthorized hosts are dropped, except packets addressed to the trusted DNS or Kerberos server; this traffic is necessary for hosts within the public subnet to authorize themselves. Once a user has authorized a host on the public subnet, the SPINACH router forwards all outgoing traffic from that particular host. An audit trail which records the identity of the user who authorized this host enables network administrators to hold the user accountable for any malicious traffic that originates from this host.

The SPINACH router forwards all *incoming* traffic, because we are solely concerned with hosts inside the prisonwall wreaking havoc upon the rest of the network, rather than the reverse. Information coming into the prisonwall from outside is not considered a security threat, because it is assumed that any hosts inside the prisonwall that are trying to extract secret information from outside machines would have to initiate such transactions from within the prisonwall, and unauthorized hosts are not allowed to send outgoing traffic in the first place.

The SPINACH router exerts no control whatsoever over *internal* traffic; these packets are carried directly from one public port to another through the LAN which connects them. Thus, any hosts that are connected inside the prisonwall must tolerate a hostile network environment.

In addition to policies regarding the awarding of network access to users, there must be policies regarding the removal of network access. At present, the SPINACH router authorizes network access for four hours at a time; the length of this timeout is a parameter we plan to experiment with, as described in Section 7. If a user wants to remain connected to the network for longer than this period, he must re-authorize his connection using the procedure described in Section 3.2.

2.3 Types of Users

In many SPINACH installations, it will be appropriate to group users according to the access permissions that should be granted to them, as well as the resources that are available to authenticate them. In our prototype installation here in Stanford's Computer Science Department, we have identified three such types of users: "Department Users," "University Users," and "Guests."

Department Users already have access to the building network in private offices and labs, but desire to connect temporarily in another part of the building, for example, to check e-mail while sitting in a conference room or lounge. Since they already have access to the building network, but simply want to connect in a different physical location for convenience, we should have no security concerns about allowing them to connect through public ports. Also, Department Users already have authentication records in the campuswide SUID (Stanford University Identification) database.

University Users already have access to Stanford's computer network in the public computer labs, and perhaps in the residence halls, but do not presently have the ability to connect to the network within the Computer Science building. System administrators within the CS Department are rightfully concerned about allowing them unrestricted access to networks within our building that they have not been able to use in the past. Like Department Users, University Users already have entries in the SUID database.

Guests are not in the SUID database and thus do not currently have the ability to access Stanford's network at all. Typically this group contains visitors from industry and other universities who are in the CS Department to meet with professors and students or attend symposia. Quite often these visitors bring their own laptop computers and would like to connect to their home networks through the Internet to access their e-mail or retrieve files. Before the implementation of the SPINACH system, there was no established mechanism for allowing these short-term visitors network resources, so guests have been forced to use low-bandwidth, high-cost wireless connections or informally borrow the use of a desktop machine in some willing person's office. Because relationships with these outsiders are important to Stanford, we should provide a mechanism for them to utilize our network resources in some reasonable way while they are visiting.

In general, different types of users may be extended different access rights on the network, at the discretion of the network administrator. In our case, due to the concerns of department network administrators, University Users are currently denied network access; Department and authorized Guest users are allowed unrestricted network access.

2.4 Hardware and Software Requirements of the Client

Especially because we have the various classes of users described above, it is important that we support many different configurations of hosts with minimal assumptions about the software present on these machines. Even University and Department users have a variety of platforms: DOS, Windows 3.1, Windows 95, Macintosh, and various flavors of Unix. We cannot foresee all platforms visitors from off-campus will use. Thus, writing and maintaining special network access software for such a large and growing number of platforms would be a burden on our network administrators. Also, visiting users would need to install this custom software on their computers to use our system, and that could be a hassle for them. We would thus like to rely solely on client software that most users will already have installed on their networked computers, or can easily obtain from other sources.

We *can* assume that the user's computer has some basic network software on it, since the user presumably has been using it to connect to some other network. Almost all networked computers will have either a telnet client or a web browser; if a visitor's computer has neither of these, they can most likely obtain one easily from a number of sources. (Our prototype system requires users to run a telnet client; an alternative web interface is currently under construction.) In addition, an increasing number of networked computers have Dynamic Host Configuration Protocol [3] (DHCP) and/or Kerberos [8] clients—for example, the widely-used Windows 95 operating system includes DHCP client software. In the design of our access restriction system, we require only a telnet client on the visitor's computer; if a DHCP or Kerberos client is present, we use it to simplify the configuration and authorization processes.

2.5 Requirements of the Network Infrastructure

Although it is less of a concern than the minimal software requirements on the client end, we also want to minimize the amount of maintenance overhead on the SPINACH router and elsewhere in the network. The less of a burden we place on network administrators, the less resistance we will encounter in deploying our system both within our department and in other institutions.

We take advantage of the existing campuswide Kerberos authentication service, as well as the departmental DNS server, to simplify some users' connection process as described in Section 3.2. No modifications to these servers are required. The only modification required of the network infrastructure beyond the SPINACH router itself is that the Department Router (see Figure 1) must be configured to forward all packets destined for the public subnet through the SPINACH router.

3. User Interfaces

Most users only see the client interface to SPINACH, through which they enable network access across the SPINACH router. Both this interface, and the interface used by network administrators to maintain the SPINACH system, are described in this section.

3.1 Authentication Mechanisms

To limit use of the public ports to those who are authorized, SPINACH must provide mechanisms for users to authenticate themselves when connecting their hosts to the public subnet. Once the user's identity has been proven, the SPINACH router can enforce the security policies listed in Section 2. Until the user authenticates himself, he must be treated as an unauthorized user.

Department and University users have permanent entries in the campuswide Kerberos authentication database. By installing Kerberos client software on their laptops, these users may authenticate themselves with the SPINACH router by presenting a Kerberos ticket that has been obtained from the campus server. Department and University users who do not install Kerberos software on their laptops may obtain one or more guest passwords and connect using the same method as Guests (described below).

To become an authorized Guest user, a visitor must obtain clearance to use our network facilities. This is provided in the form of a *guest (userID, password) pair* which is generated at the request of some authorized person such as a faculty member or network administrator (see Section 3.3). The userID and password are both human-readable strings which can be given to the Guest and which the Guest enters using a telnet or HTTP connection to the SPINACH router. Since this transmission goes across the public subnet in cleartext, it must be a single-use password so that replay attacks are fruitless [6]. The userID need not be unique; it simply makes the system more robust in the face of password-guessing attacks, by increasing the number of combinations that must be attempted by miscreants.

3.2 Connection Procedures

A user who is connecting his laptop to a public network port within a SPINACH installation follows these steps:

1. The user connects his laptop to one of the public Ethernet ports.
2. If the user's laptop has DHCP client software, it automatically retrieves network configuration information from a DHCP server running on the SPINACH router and configures the laptop accordingly. Since this exchange occurs between the laptop and the SPINACH router—entirely *within* the public subnet, rather than *through* the prisonwall—packets from the as-yet unauthorized laptop are not blocked. If the user's laptop does *not* have DHCP client software, the user must configure the laptop's network software manually, entering the IP address and IP routing information marked on the Ethernet port so that packets are properly routed through the SPINACH router.
3. If the user is permanently authorized—that is, a Department or University User—and has Kerberos client software on his laptop, he enters his personal password into his Kerberos client software to obtain a *ticket* from the trusted campus Kerberos server (see Figure 1). A special IP packet filter rule on the SPINACH router allows unauthorized machines to communicate with only the trusted campus Kerberos server and the department's DNS server through the prisonwall. If a permanently authorized user wants to access the network from a laptop *without* Kerberos client software, he must obtain a one-time guest password and log in in the same way visitors do (described in the following step).
4. The user initiates a telnet connection to the SPINACH router using the telnet client on his laptop. As in step 2, since this communication is *to* rather than *through* the prisonwall, it is not blocked. If the telnet server sees that the user has obtained an appropriate ticket from the trusted campus Kerberos server, the IP address and hardware (Ethernet) address of the laptop are recorded and the laptop is authorized to use the network facilities. Otherwise, the modified telnet server on the SPINACH router prompts the user to enter a userID and single-use guest password. If the user enters a valid (userID, password) pair, network access is granted.

Once the SPINACH router's modified telnet server has granted network access, a filtering rule (as described in Section 4.3) is added that allows all traffic coming from this host to be forwarded out of the prisonwall as necessary for a certain length of time. The user's telnet client displays a message to this effect and then is automatically disconnected. The user then has unrestricted network access for a certain length of time (currently four hours).

3.3 Generating Guest Passwords

We must provide some mechanism for generating guest passwords for visitors. These guest passwords are generated on the SPINACH router itself; this avoids transmitting them in cleartext across the network until they are actually used. A small number of users, chosen by the network administrators who install the SPINACH system, are given user accounts on the SPINACH router. When these users initiate a Kerberos-authenticated and encrypted telnet session with the SPINACH router, they are allowed to log in and obtain a shell process on the router (which is running Unix). Then they can run a special password-generating program that creates human-readable one-time passwords. The (userID, password) pairs are entered into a database on the router for future comparison and displayed on the user's telnet client. Since the telnet session is known to be encrypted, there is no danger of new passwords being snooped by other hosts on the network. It is up to administrators at sites where SPINACH is deployed to develop a mechanism for distributing the one-time passwords to visiting users.

3.4 Long-Term Maintenance

One of the goals of the SPINACH system is to minimize the maintenance required for continued operation. Besides generating guest passwords, there is usually no manual maintenance required. But, should a network administrator want to examine the audit trail maintained by the SPINACH router or debug a problem on the router, he can follow the same procedure as mentioned above to log in and execute arbitrary commands on the router. The number of users with accounts on the SPINACH router should be kept to a minimum so that there is less chance of malicious or inept activity on the router that opens security holes.

4. System Implementation

4.1 Software on the SPINACH Router

The SPINACH router is an Intel Pentium-based computer running a Linux 2.0.30 kernel modified to filter IP packets based on hardware address as well as IP address. Since the SPINACH router is connected to a network with many untrusted hosts, it is best to run as few network servers as possible on the router to reduce the possibility of break-ins [2][4]. However, there are a few pieces of software that must be running on the router to implement prisonwall functionality—that is, to forward network traffic into and out of the prisonwall (subject to the policies in Section 2.2), and to allow hosts within the prisonwall to move from "unauthorized" to "authorized" status as appropriate.

The following pieces of software must be running on the SPINACH router:

1. **packet filter**: routines within the IP forwarding code in the kernel that allow packets to be routed or dumped selectively, based on source and destination port numbers and IP addresses, as well as source hardware addresses.
2. **prisonguard**: a user-level process that is always running on the router, modifying the packet filter parameters in the kernel as necessary and maintaining databases of guest passwords and authorized University/Department users.
3. **modified telnet server**: modified so that when most users connect to the telnet server on the SPINACH router and are properly authenticated, network access is enabled but a login shell is not provided.
4. **authorization clients**: processes that communicate with the prisonguard process to enable and disable network access and generate guest passwords.
5. **DHCP server**: a standard DHCP server, with no authentication extensions, which allows for automatic configuration of IP and higher-layer protocol information on any host with a DHCP client.

4.2 Communication Between Authorization Clients and the Prisonguard

The *prisonguard* process is so named because it maintains control over all security features of the prisonwall (SPINACH) router. It keeps a record of all generated guest (userID, password) pairs as well as a list of permanently authorized users, validates entered guest passwords against the list of generated ones, and modifies the packet forwarding rules in the kernel as appropriate in various cases. While other pieces of the software, such as the modified telnet server and the guest password management program, are short-lived processes and exist only long enough to collect information from one particular host as it moves from authorized to unauthorized, the prisonguard process runs constantly and maintains all the state necessary to perform appropriately. For example, when a host has been authorized for a certain amount of time, the prisonguard keeps track of how long the host has been operating and when its authorization should be revoked and is responsible for revoking the authorization at that time. Although the prisonguard process maintains this state in main memory for efficient operation, it also writes the information to disk periodically in case the SPINACH router crashes or shuts down.

The short-lived processes that communicate with the prisonguard and tell it to authorize and unauthorize hosts, as well as generate and check guest passwords, are called *authorization clients* (see Figure 2). They are started as a result of a new user attempting to authorize his own host, or a SPINACH administrator running code on the router itself. Only known authorization clients on the SPINACH router have a legitimate need to communicate with the prisonguard, which acts as the authorization server, so communication with the prisonguard takes place over Inter-Process Communication (IPC) [9] that is only accessible to these processes and the superuser. To implement this secure IPC, the prisonguard uses a Unix domain socket linked to a Unix file that is only accessible by the superuser and other members of the group *authclients*. The following functions are available for authorization clients to call for communication with the prisonguard:

- **guard_authorize_nonguest(ipaddr, username)**
  The Department or University user by this username (already authenticated via Kerberos) has requested use of the network. If this username is that of an approved user, allow the host with this IP address use of the network. Make note of the source Ethernet address of the last packet received from this IP address so that other hosts cannot mistakenly or maliciously assume the same IP address and pass as the same host.
- **guard_authorize_guest(ipaddr, userID, passwd)**
  A Guest has entered this userID and one-time guest password to request use of network facilities. If this (userID, password) pair is valid, allow the host using this IP address access to the network. Make note of the source Ethernet address of the last packet received from this IP address so that other hosts cannot mistakenly or maliciously assume the same IP address and pass as the same host.
- **guard_unauthorize(ipaddr)**
  Disallow the host using this IP address access to the network. As noted above, the prisonguard process takes responsibility for automatically unauthorizing

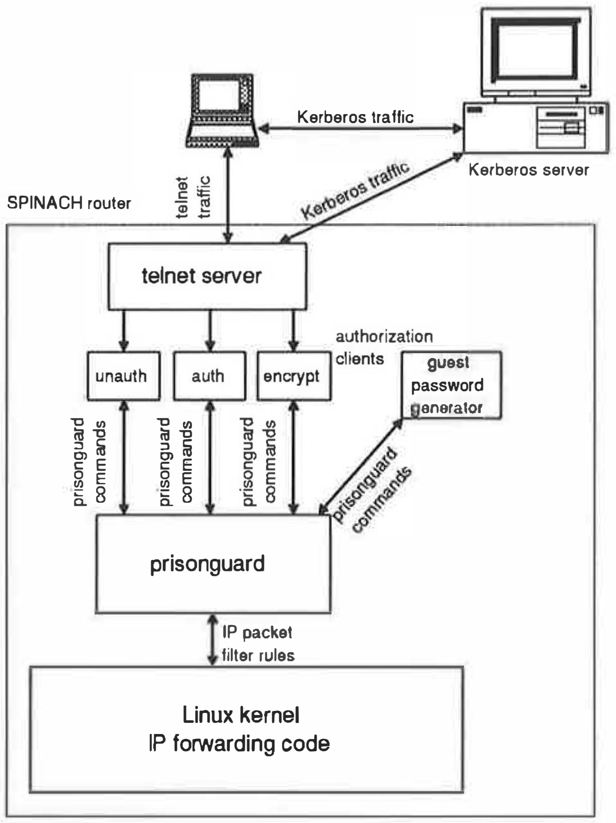


FIGURE 2. Inter-Process Communication between authorization clients and the prisonguard, and between the prisonguard and kernel IP forwarding code.

a host after some amount of time, so this command is not commonly called. However, future security improvements to the SPINACH system may require this functionality. For example, we may want to provide a tool with which network administrators can remove users' network access privileges before they would normally expire.

- **guard_getpassword(keyinfo, passwd)**
  Asks the prisonguard to generate and return a new one-time guest password. The keyinfo field contains the guest userID, as well as information about the Guest user and/or the Department contact who has requested this password, for auditing and/or billing purposes.

As shown above, the authorization clients can demand that the prisonguard allow or disallow network access to a particular host; thus, it is very important that access to all processes running with a group id of *authclients* is closely guarded. It is for this reason that only a very few users—those who are maintaining the SPINACH system or generating guest passwords—are allowed to log in and execute arbitrary code on the SPINACH router. The modified telnet server allows other users only to enable network access for their hosts.

4.3 Filtering Rules

Packets are filtered during the forwarding process within the Linux kernel [10] according to the security policies set out in Section 2. All packets leaving through the SPINACH router are assumed guilty until proven innocent; that is, they are dropped unless they meet at least one of the following criteria:

- They are destined for the trusted departmental DNS server (may be needed to find Kerberos server), *or*
- They are destined for the trusted Kerberos server (needed for authorization process), *or*
- They come from an authorized user.

A packet is judged to come from an authorized user if:

- Its source IP address is that of an authorized host, *and*
- Its source hardware (Ethernet) address matches the hardware address recorded when this host was authorized.

The second check is necessary to guard against unauthorized hosts within the prisonwall that spoof IP addresses of authorized hosts. When a host is newly authorized, the SPINACH router freezes its ARP entry that maps the host's IP address to its observed hardware address. As long as this host is authorized, the ARP entry will not be modified. Any packets that later purport to come from this host's IP address will be checked against the previously recorded hardware address. Unfortunately, hardware addresses are spoofable as well, although not as easily as IP addresses; we address this problem in the following section.

5. Security Considerations

Any network security system involves tradeoffs among the strength of security provided, the equipment required for implementation, and the inconvenience caused to its users. Network administrators must decide which factors are most important and choose a security system accordingly.

The SPINACH system is designed with minimal assumptions about the network hardware and client software. We have made these constraints so the system will be useful in many network installations, and to free network administrators from the burden of maintaining special software for an ever-changing array of client machines. The cost of these gains is that the SPINACH system does not provide the same level of security as some other schemes (two of which are described in the following section) that require more sophisticated network hardware or client software.

By filtering packets from within the prisonwall by hardware address as well as IP address, the SPINACH router prevents casual miscreants from using public network ports without authorization. It is fairly simple in most operating systems to modify the source IP address of transmissions manually; modifying the source hardware address is more difficult, making illicit use of the network that much less likely. The determined hacker, however, will be able to obtain unauthorized access to the network by observing traffic on the public subnet, capturing hardware address / IP address pairs from legitimate users, and then modifying his network interface parameters to imitate an authorized user. Depending on the particular situation, the legitimate user may or may not be aware that something is amiss.

In practical terms, there are two types of Ethernet media: switched and shared. In a shared medium, hardware address filtering is the best security we can provide without additional software requirements of the clients. On a switched Ethernet, as we have in our building, packets that are sent to or from a particular host are not seen at all network ports; the switch routes packets destined for a particular hardware address to its port only. Because of this, it is significantly harder in this type of system for a malicious user to snoop other users' hardware addresses and then impersonate them. It is still possible, however; broadcast packets are typically forwarded to all network ports, at which point the source hardware address can be discovered by a malicious user.

By requiring a switched Ethernet instead of the much cheaper (and often already installed) shared-medium variety, the SPINACH router could communicate with the LAN switch to associate a physical port identifier with each legitimate user and thus prevent a hacker on some other port from imitating the legitimate user. Another approach would be to require custom software on all clients so that they could be repeatedly authenticated without user intervention. The first approach might require an expensive retrofit of the existing network architecture; the latter would require special software to be installed on all clients. If the network administrators require absolute protection against unauthorized network use at any cost, these changes may be appropriate; however, if they are constrained by time and money, they can use SPINACH to provide the best security possible for their network.

SPINACH is also vulnerable to a denial-of-service attack mounted by a malicious user against the Kerberos or DNS servers. Such an attack could block access to those services even to people outside the public network. A seemingly simple solution to this problem is to filter out Kerberos and DNS requests in addition to other packets from sources that seem to generate an extraordinary number of requests. With more experience about normal request loads, we will be able to judge whether this is a reasonable approach.

Another security concern in the SPINACH system is the vulnerability of hosts connected *within* the public subnet. Since all filtering by the SPINACH router occurs on the boundary of the public subnet, rather than within it, there are no restraints placed on the interaction between hosts on the public subnet. This means that even unauthorized users connected to the public subnet will be able to send packets to and receive packets from authorized users on the same subnet. Users of the public subnet should be made aware of this possibility; any user of the public subnet uses the network at his own risk, and it is his responsibility to encrypt his transmissions or fortify his host against attack.

Many network administrators may find that installing a system such as SPINACH will keep unauthorized network use to a tolerable level. SPINACH does not preclude the use of traditional firewalls; installing firewalls around the most sensitive parts of a network will protect them from abuse from public network ports and elsewhere.

6. Related Work

We are aware of two other proposals to deal with the problem we attempt to solve with the SPINACH system: Carnegie Mellon's NetBar proposal, and UC Berkeley's position paper, "Authenticating Aperiodic Connections to the Campus Network." Both of these systems, rather than using hardware address information to filter packets, use expensive, proprietary solutions at the link layer to isolate unauthorized hosts. Also, they both require more in terms of software that must be installed on every client.

In the NetBar system [7], a Cisco Catalyst VLAN switch is used to isolate unauthorized hosts. When a new client attaches to an available network port, it receives an IP address via DHCP, and the port is connected to a VLAN with limited connectivity suitable only for completing the authorization process. The user then uses Kerberos to authenticate himself, at which point the NetBar server sends an SNMP message to the VLAN switch, connecting this port to an "attached" network with full connectivity. Using the VLAN switch to connect and disconnect hosts in this manner provides more security than SPINACH, because it is not vulnerable to hardware address spoofing. However, it also uses proprietary signaling methods and thus relies on the presence of a specific brand of VLAN switch. (Many existing Ethernet installations do not use a VLAN switch at all; the NetBar's designers admit that the need for switched Ethernet ports can often constitute a financial barrier, particularly on a college campus.) In addition, the NetBar proposal requires all hosts to run DHCP and Kerberos client software, and only allows users with entries in the Kerberos database to connect to the network. The NetBar could be easily extended, though, to allow the use of guest passwords as we have done in SPINACH.

UC Berkeley's proposal [11] is even more demanding in terms of client software and network infrastructure. Hosts that are connecting to the campus network must contain DHCP client software that has been enhanced to exchange authentication information, rather than using the widely implemented Kerberos protocol or allowing the entry of guest passwords over telnet. This makes it extremely difficult for short-term visitors to make use of the campus network; they must install new DHCP client software on their machines, and network administrators must maintain such DHCP clients for all platforms that visitors are allowed to use. Also, their scheme for enabling and disabling communications on particular network ports, while more secure than hardware address filtering, requires the use of a LAN hub that has been specially modified by the manufacturer. This will make the use of this system impossible in many cases.

7. Future Work

As we receive feedback from users and administrators of the SPINACH system in our building, we will modify the policies and interfaces to fit their suggestions. We anticipate adding a more user-friendly HTTP interface for Guest users to authenticate themselves. Also, we may implement the S/IDENT protocol as an alternative to Kerberized login for authentication of Department and University users that would require no manual login procedure.

In the interest of improving security without requiring more sophisticated network hardware and client software, we will also tune the security policies to minimize a hacker's "window of opportunity" obtained through hardware address spoofing. First, we will adjust the authorization timeout period to reflect the observed duration of network usage, so that it is long enough to inconvenience few users but limits the amount of time a hacker can access the network with a captured hardware address after the legitimate user has stopped using the network. Also, we will look into disabling network access for a host after some period of inactivity, the goal being to mark an IP address / hardware address pair as unauthorized before a hacker realizes that the legitimate host is no longer on the network.

If we conclude from network administrators' feedback that a defense against hardware address spoofing is necessary, we will probably incorporate dynamic VLAN switching similar to that used in the NetBar, described above, to authenticate traffic based on physical LAN port rather than hardware address. This solution would significantly increase the cost of the network infrastructure needed to support SPINACH, but by combining dynamic VLAN switching with the dynamic IP filtering possible with the SPINACH router, we could continue to support client hosts without any special network software.

8. Conclusions

The Secure Public Internet Access Handler strikes a good balance between allowing temporary network users the freedom to use whatever applications they like on whatever platforms they like, and limiting network access to authorized users. Because it does not limit users to using a prescribed set of applications and protocols, it does not prevent users with network access from causing trouble. Instead, it limits the pool of users to those who will be accountable for their actions, i.e., Stanford affiliates and identified guests. This is an appropriate solution for an academic research environment, and may also be the best solution for public organizations such as schools and libraries that require some control over who is using their networks without the large capital outlays needed for alternative solutions.

As hackers become more savvy and switched Ethernet hubs less expensive, more feature-rich, and, we hope, more standardized, it may well make sense to use a switched LAN architecture to isolate unauthorized network users. Until this day comes, however, SPINACH will provide a useful measure of access control even in shared-medium Ethernet installations, and even better (though not bulletproof) security in a switched Ethernet environment.

Source code and other information regarding SPINACH will be made freely available on the MosquitoNet Project web site, http://mosquitonet.stanford.edu/mosquitonet.html.

9. Acknowledgements

We would like to thank fellow group members Stuart Cheshire, Kevin Lai, Petros Maniatis, Diane Tang, Akihiro Tominaga, and Xinhua Zhao for their help in improving both the design of SPINACH and its presentation in this paper. We particularly thank Mema Roussopoulos for her help in implementing the system as well.

In addition, we thank Dan Boneh and Jonathan Stone for their advice in the design of SPINACH, and Armando Fox, Darrell Long, Nick McKeown, Lorenzo O'Reilly, and the anonymous reviewers for their careful readings and help in improving this paper.

This work was supported in part by a Terman Fellowship and a grant from the Keio Research Institute at SFC, Keio University, and the Information Technology Promotion Agency, Japan.

10. References

1. Alexander, S. and R. Droms. "DHCP Options and BOOTP Vendor Extensions; RFC-2132." *Internet Request for Comments*, no. 2132, March 1997. <http://ds.internic.net/rfc/rfc2132.txt>

2. Cheswick, William R., and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, Reading, MA, 1994.

3. Droms, R. "Dynamic Host Configuration Protocol; RFC-2131." *Internet Request for Comments*, no. 2131, March 1997. <http://ds.internic.net/rfc/rfc2131.txt>

4. Garfinkel, Simson, and Gene Spafford. *Practical Unix Security.* O'Reilly & Associates, Sebastopol, CA, 1991.

5. Greenwald, Michael B., Sandeep K. Singhal, Jonathan R. Stone, and David R. Cheriton. "Designing an Academic Firewall: Policy, Practice, and Experience With SURF." *Proc. IEEE Symposium on Network and Distributed Systems Security*, 1996. <http://www-dsg.stanford.edu/papers/firewall/>

6. Haller, Neil M. "The S/Key One-Time Password System." *Proc. Internet Society Symposium on Network and Distributed System Security*, pp 151-157. San Diego, CA, February 1994.

7. Napjus, Erikas. "NetBar: Carnegie Mellon's Solution to Authenticated Access for Mobile Machines." <http://www.net.cmu.edu/design/netbar.html>

8. Steiner, J. G., C. Neuman, and J. I. Schiller. "Kerberos: An Authentication Service for Open Network Systems." *Proc. USENIX Winter Conference*, February 1988, pp 191-202.

9. Stevens, W. Richard. *Unix Network Programming*. PTR Prentice Hall, Englewood Cliffs, NJ, 1990.

10. Vos, Jos, and Willy Konijnenberg. "Linux Firewall Facilities for Kernel-Level Packet Screening." *Proc. NLUUG Spring Conference 1996*. De Reehorst, Ede, The Netherlands, May 1996. <http://www.xos.nl/linux/ipfwadm/paper/>

11. Wasley, D. L. "Authenticating Aperiodic Connections to the Campus Network." *ConneXions* (Interop August 1996), vol. 10, no. 8, pp 20-26. <http://www.ucop.edu/irc/wp/wp_Reports/wpr005/index.html>

Web Facts and Fantasy

Stephen Manley, *Network Appliance*
Margo Seltzer, *Harvard University*

Abstract

There is a great deal of research about improving Web server performance and building better, faster servers, but little research in characterizing servers and the load imposed upon them. While some tremendously popular and busy sites, such as netscape.com, playboy.com, and altavista.com, receive several million hits per day, most servers are never subjected to loads of this magnitude. This paper presents the analysis of internet Web server logs for a variety of different types of sites. We present a taxonomy of the different types of Web sites and characterize their access patterns and, more importantly, their growth. We then use our server logs to address some common perceptions about the Web. We show that, on a variety of sites, contrary to popular belief, the use of CGI does not appear to be increasing and that long latencies are not necessarily due to server loading. We then show that, as expected, persistent connections are generally useful, but that dynamic time-out intervals may be unnecessarily complex and that allowing multiple persistent connections per client may actually hinder resource utilization compared to allowing only a single persistent connection.

1. Introduction

The public's enthusiasm for the Web has been matched only by that of computer companies. Since money drives the industry, the emphasis from Microsoft to the smallest start-up has been on rapid technological development, rather than well-reasoned scientific advancement. New Web tools make money, statistical analyses of the Web do not. Concurrently, the academic community is often constrained by lack of data from sites outside the academic world [1][2][6]. As a result, most existing analyses are outdated. Most published statistics on server behavior come from data gathered in late 1994. In the past three years, the nature of the Web has fundamentally changed. CGI has introduced server and user interaction. Java and animated GIF images have continued to find their way onto Web pages. The number of Web users has grown at an unknown, but predictably exponential rate. With the tremendous growth and change in Web sites, users, and technology, a comprehensive analysis of real traffic can help focus research on the most instrumental issues facing the computing community.

The statistical analysis presented in this paper focuses on traffic patterns observed on a variety of Internet Web sites (intranet servers have been omitted from this study due to the unavailability of intranet server logs). Server logs reveal an enormous amount of information about users, server behavior, changes in sites, and potential benefits of new technical developments. In order to design next generation services and protocols (e.g., HTTP-NG), it is crucial to understand what the Web looks like today, how it is growing, and why it is growing as it is. In this paper, we use internet Web server log analysis to dispel and confirm widely held conceptions about the Web. Section 2 describes the initial set of sites we surveyed to derive our Web site taxonomy and outlines the basic growth characteristics of these sites. Section 3 presents a simple taxonomy for describing site growth, dispelling the myth that all Web sites are alike. Section 4 dispels the myth that CGI traffic is becoming uniformly more important. Section 5 addresses the issues surrounding persistent connections and how to maximize the benefit derived from them. Section 6 takes a first step towards answering the question, "What makes users wait?" by showing that servers are not necessarily the primary source of latency on the Web. Section 7 concludes.

2. Site Survey

Our Web log analysis is based on server logs obtained from a variety of sites. The sites were chosen to cover a broad range of topics, user populations, popularity, and size. Due to our agreements with several of our providers, we are unable to identify the sites in question, so we provide descriptions of the sites instead. Table 1 summarizes our initial set of server logs.

The sites in our survey can be broadly described in three basic categories: academic sites (EECS, FAS, ECE), business (BUS, ISP, AE, WEB, FSS), and informational (PROF, GOV). Within these categories, the sites exhibit different characteristics. For example, the business sites represent each of the major business models on the Web. ISP is an internet service provider that advertises its services. BUS also uses the Web for advertising, but its business has nothing to do with the Web. AE comes from the set of ubiquitous adult-

Abbr	Anonymized Site Name	Site Type	Content	Service Provider	Server Software	Time
BUS	Traditional Business	.com	Information on subject matter, advertisements	ISP	Apache 1.1.3	1/96 - 2/97
EECS	Harvard University Electrical Engineering and Computer Science	.edu	Graduate student Web pages, department information	Harvard EECS	NCSA 1.4.2	4/96-2/97
FAS	Harvard University Faculty Arts and Sciences	.edu	Information for an academic institution, student Web pages	Harvard FAS	NCSA 1.4.2 Apache 1.1.3	10/94-2/96
ISP	ISP company page	.com	Simple advertisement	ISP	Apache 1.1.3	9/96 - 2/97
ECE	Rice University Electrical and Computer Engineering	.edu	Graduate student Web pages, department information	Rice ECE	Netscape Netsite-Commerce 1.0	7/95-12/96
AE	Adult-Entertainment	.com	Adult images, movies, chatrooms	ISP	Apache 1.1.3	3/96-9/96
PROF	Organization for Members of same Profession	.org	Articles and images pertaining to field	ISP	Microsoft IIS/3.0	4/96-2/97
WEB	Web site designer	.com	Samples of different sites, games	ISP	Apache 1.1.3	8/96-2/97
GOV	Government Agency	.gov	Information on agency's actions	ISP	Apache 1.1.3	8/96-2/97
FSS	Free Web Software Site	.com	Evaluation copy of proprietary Web software	ISP	Apache 1.1.3	4/96-1/97

Table 1: Site Survey Description. The educational sites (EECS, FAS, and ECE) differ from the rest of the sites in that the sites' content is not the uniform product of a single webmaster, but instead a conglomerate of a number of independent Web publishers.

entertainment sites. FSS makes its living by licensing a Web software product, but allowing visitors to download a less functional version of the product for free. The final business model is that of WEB, which uses its site as an advertisement for a Web product. Although all the ISP logs in Table 1 come from a single ISP, we have analyzed logs from other providers and found that these sites are indicative of the other providers' sites as well.

Unsurprisingly, the characteristic common to nearly all our sites is an exponential rate of change. Table 2 shows this change in requests, bytes transferred, number of files on the sites and the number of bytes on the site for each of our surveyed sites. While the derivative of the change for three of our sites is negative, perhaps the most astounding result is that even the slowest growing sites double each year and our fastest growing site doubles each month.

3. A Web Site Taxonomy

Much Web research tends to assume that all interesting sites have traffic loads similar to those of Microsoft and Netscape. These sites each have more than ten servers to handle tens of millions of requests each day, claiming to be two of the most popular on the Web. However, as the most heavily loaded sites, they cannot also be the common case. While most of the sites in Table 2 demonstrate substantial growth, the loads, shown in Table 3, vary tremendously. A site handling fifteen million requests for seventy-six GB of data per month (FSS) must be thought of differently than a site processing forty-five thousand requests for 250 MB of data per month. Comparing these sites directly is unlikely to yield very interesting results. Load is only one way in which sites differ; the size of the site, the diversity of the population that is attracted to the site, the growth patterns, the user access patterns, and how the site changes all play large roles in characterizing a Web site. From our log analysis, we have concluded that the three primary issues that characterize a site are: site composition and growth, growth in traffic, and user access patterns. While the data for the first two factors can be found in Table 2 and Table 3, user access patterns are not easily described. The distribution of requests per file and distribution of number of requests per user indicates whether users tend to visit many pages on a site, or only a few. These figures also indicate whether all users visit the same subset of pages, or tend to view different subsets of pages on the site. We present a more detailed analysis of these phenomena in earlier work [7].

Site	% Growth per month for duration of logs.				Double (Half) Interval
	Reqs	Bytes	Files	Bytes	
			on Site		
Traditional Business	60	105	37	67	2 months
Harvard EECS	28	18	19	16	3 months
Harvard FAS	27	31	33	33	3 months
ISP	-2	7	-2	9	3+ years
Rice ECE	13	17	7	14	6 months
Adult Entertainment	-27	-29	1	-1	3 months
Organization	-21	-20	-23	-19	3 months
Web Site Designer	6	7	0	14	1 year
Government Agency	7	5	1	-7	11 months
Free Software	95	81	23	24	1 month

Table 2: The monthly growth patterns of each site and its traffic. As the growth for nearly all these sites is exponential, the interesting question becomes, "How long does it take to double?" As we can see by the Free Software Site, there are examples of sites that nearly double every month while other sites (Web Site Designer) grow more slowly. Some of the sites actually demonstrate negative growth, another frequent Web phenomenon that will be discussed in Section 3. In particular, the Adult Entertainment site no longer exists. The reported data traces its reduction to destruction.

Site Name	Requests	Monthly Transfer Rate (MB)	Files on Site	MB on Site
Traditional Business	321,747	3,819	347	2.8
Harvard EECS	106,001	1,322	5,865	196.0
Harvard FAS	2,328,401	15,097	34, 348	455.0
ISP	8,139	39	134	1.5
Rice ECE	85,763	854	4,655	115.0
Adult Content	69,906	857	223	5.5
Organization	42,301	251	95	0.8
Web Site Designer	43,523	104	119	0.7
Government Agency	26,049	214	185	1.2
Free Software	15,982,085	76,315	4070	136.0

Table 3: The size of sites and the traffic handled in the most recent server log (one month). The disparity in levels of traffic and site size illustrate the fundamental difference in Web sites.

During the course of our monitoring of these sites, we visited each site frequently to determine how the sites were evolving and then used that information in conjunction with the logs to discern basic trends in site growth. We conducted a regression analysis on the growth of the site (as measured by the number of requests) for every parameter we could measure. We found that many parameters appeared to have a slight influence on growth, but we focused on that parameter that correlated most closely with growth. For some sites, the parameter attributed to growth showed excellent correlation (e.g., better than 95% confidence intervals for sites such as AE). For other sites, the best parameter produced 80% confidence intervals (e.g., FSS). In all cases, the best parameter provided confidence intervals of at least 80%. Table 4 summarizes the growth patterns.

Growth Function	Site(s)	Explanation
# of Web Users	Single topic sites: FSS	More users learn of the site and visit it to download software.
Site overhaul	Aggressive business advertising: BUS	Grow in bursts as the webmasters "renovate" the site.
Number of documents on site	Academic sites: EECS, FAS, ECE	Have a disproportionate number of pages, and their popularity increases as more users create pages on the site.
Documents visited per user	Non-aggressive businesses or special interest sites: ISP, GOV, WEB	Lure visitors into visiting more of the site as it develops.
Number of search engine hits	Competitive markets: AE	Grow based on number of times the popular search engines find them.
Cost	Pay-for-View Sites: PROF	Increasing fees are met with decreasing traffic

Table 4: Characteristics that Categorize a Web Site. The "growth function" column identifies the parameter that most closely correlates to a site's growth. We hypothesize that FSS is representative of a very large class of sites whose popularity grows with the user population of the Web. Search engine sites and sites for general entertainment and information (CNN, ESPN, etc.) are hypothesized to fall in this category as well.

There are a variety of ways in which sites can grow. They may grow because new users are drawn to the site or because existing users visit more frequently or more deeply. Our first class grows by attracting more visitors to the site, and we speculate that the number of visitors is a function of the total Web user population. The free software site has a singular, wildly popular product. As more people learn of the software, more people visit the site to download the software. The accesses on this site are heavily skewed: 2% of the documents account for 95% of the site's traffic.

A second growth model is to explicitly renovate a site in an attempt to increase traffic. The business using the Web to market aggressively (BUS) demonstrates this

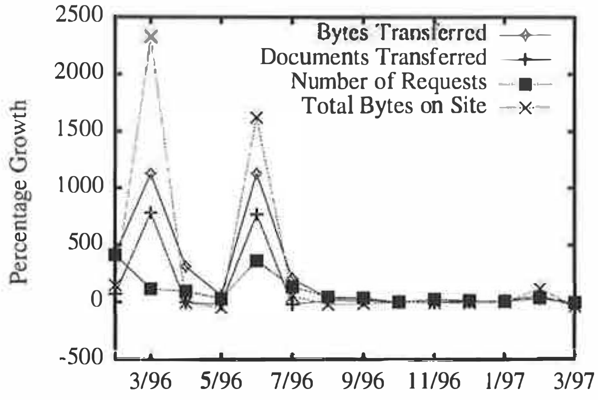


Figure 1: Traditional Business Site Growth. The two dramatic peaks indicate periods where the site was overhauled. Each time, the site's traffic exhibited a significant increase and then leveled off at this new level.

growth. Figure 1 depicts the site's growth during our evaluation interval. All growth occurs in bursts, whose timing corresponds to major reorganizations of the site. The first burst corresponds to a shift from a text-based site to a graphics-based site. The second burst corresponds to adding more depth to the site, adding more details about specific products and catering to particular classes of customers (e.g., women or young adults). After each reorganization, the site undergoes tremendous growth, which tapers off, and levels out at a volume that is significantly greater than it was before the renovation.

The third category is typified by the academic sites. The content of these sites is not controlled by a single Web master. Instead, it tends to grow with the user population; as the site grows in size, so do the number of requests to that site.

The fourth classification contains those sites whose traffic increases by attracting users to visit more of the site. There are two discernible patterns that characterize these sites. First, the number of requests closely tracks the number of documents on the site. Second, the average number of pages visited per session also tracks the growth of the site, as shown for the GOV site in Figure 2. The sites rarely change scope, but additional material is added on specific subjects, and the visitors respond by viewing a larger fraction of the site.

The final two classes exhibit negative growth. First, consider the case of the Adult Entertainment site, which no longer exists. The business model of the site is like many on the Web—the user is given access to a limited subset of free material followed by a request for payment to get access to the remaining material. With the tremendous growth of the Web, and almost

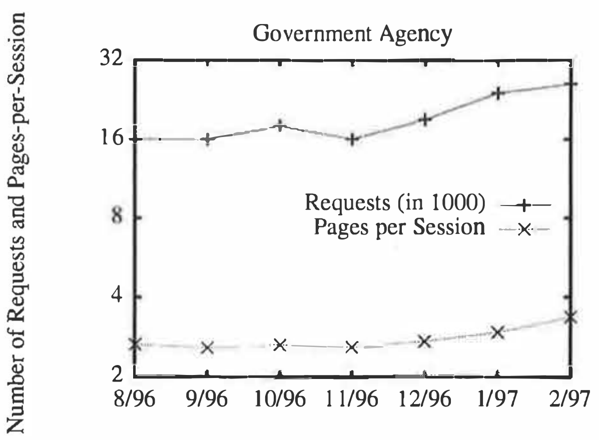


Figure 2: Growth tracks the number of pages visited per session. This shows the Government Agency where the gentle growth in number of requests stems from a similar growth in the number of pages visited by each user.

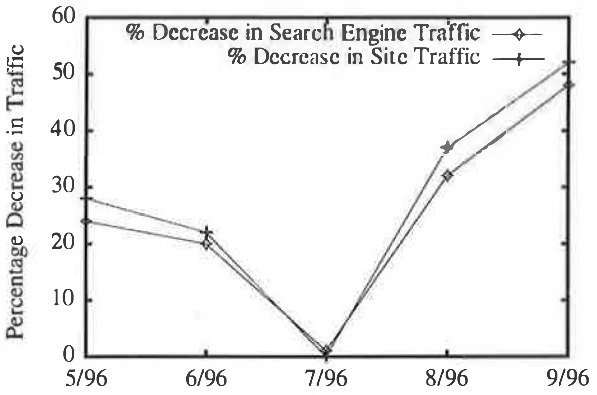


Figure 3: Growth of Search Engine Hits and User Requests for the Adult Entertainment Site. Those sites in a competitive market, such as the adult entertainment market, can live or die by their ranking in the various Web search engines.

ubiquitous nature of such sites the best means of attracting users is through the search engines. Not surprisingly, the site experienced growth of almost a factor of 60, when it began to receive requests that were traced to the most commonly used search engines (e.g., Alta Vista, Yahoo, and Excite). The site's popularity began to decrease, however, without the site or user access patterns changing drastically. The number of unique users began to drop. Figure 3 shows that the number of user requests mirrors that of the number of references from the Web's search engines. Those sites that depend on search engines for rapid growth can also suffer a rapid downfall, when the search engine does not return the site's URL as one of the best matches.

Our final category also depicts negative growth. In this case, the site's popularity dropped off by an order of magnitude as soon as it began charging for access, and then began to exhibit growth similar to that of the non-aggressive business sites; those users who stuck with the site even after the site began to charge for access increased their use of the site, viewing an increasing fraction of it over time.

After completing the taxonomy based on our first set of logs (those shown in Table 1), we analyzed a set of logs from a second ISP to verify if they too fit into our taxonomy. The second ISP's sites all fell into two of our six categories: 15 of 28 sites grew by attracting more users and 13 of the 28 grew by encouraging users to visit the site more thoroughly, thus viewing more pages per session.

There is a class of sites that have been omitted from this survey, namely the Web search engines. We hypothesize that these sites fall into our existing taxonomy with respect to growth, in that their load reflects the growth of the Web in general (as does the Free Software Site or sites whose load grows as a function of the number of different visitors). However, they do not fall within the following discussion about the importance of CGI. It is obviously the case that sites hosting search engines will be extremely sensitive to CGI performance (or whatever they use to implement searching capabilities).

4. The Effects of the Common Gateway Interface (CGI)

Because processing CGI is frequently much more computationally expensive than returning static documents, its perceived importance has motivated a great deal of server development. Both Netscape and Microsoft have changed the interface for CGI traffic, to improve performance. Microsoft's site includes two servers dedicated to processing their equivalent of CGI. Yet, with all of the clamor, the sites we surveyed derived little functionality from CGI. Table 5 shows that of the servers we surveyed, most process very little CGI traffic. In fact, only three sites report more than 2% of their traffic due to CGI.

Of the sites we surveyed, the most widely used CGI script was the ubiquitous counter (a simple CGI script that tallies the number of accesses to a particular page) and the second most frequently occurring script was the redirect, a script that indicates that a page has moved. Although CGI, in general, is often assumed to be an order of magnitude slower than returning static HTML documents, these particular instances of the use CGI require about as much processing overhead as static documents [7]. The Adult Content site, Free Software Site, and Organization site also use CGI scripts to allow users to log into the site, and the Traditional Business and FAS sites provide search engine capabilities, which are responsible for a noticeable fraction of their CGI traffic. Even so, these hits account for a tiny fraction of the traffic on all but the organizational site, which is rather unusual in that all external requests are directed through a CGI-driven interface. The other anomalous site is EECS where students have access to the CGI bin and can create their own scripts. This site exhibits the greatest diversity in CGI and explains the relatively large percentage of traffic (and bytes) due to CGI (see `http://www.eecs.harvard.edu/collider.html` for a particularly creative use of CGI scripts). Perhaps most interestingly, we find that, not only is the use of CGI fairly low across all sites, but the percentage of traffic due to CGI did not increase over the course of our measurement interval.

Site Name	%Requests as CGI	%Bytes Transferred from CGI
Traditional Business	1.0	0.4
Harvard EECS	8.0	15.0
Harvard FAS	1.4	1.6
ISP	0.0	0.0
Rice ECE	0.0	0.0
Adult Content	2.0	0.0
Organization	34.0	62.0
Web Site Designer	1.0	0.0
Government Agency	0.0	0.0
Free Software	10	5.0

Table 5: The percent of requests due to CGI. Most servers process very little CGI, and the traffic it generates accounts for a small fraction of the site's traffic.

In the logs we have examined, the latency of CGI requests has mirrored that of regular requests, and we find that sites with significantly different ratios of CGI to non-CGI requests exhibit the same latency patterns. Based on this observation and the fact that the ratio of CGI traffic to regular traffic is not changing, we conclude that the long latencies users are experiencing at these sites or any increased slowdowns of these sites is not due to CGI. Section 6 presents a more detailed discussion of observed latencies.

5. Persistent Connections

The HTTP/1.1 specification [4] calls for support of persistent connections; that is, rather than initiating a new connection for every document retrieved from a server, a long-lived connection can be used for transmitting multiple documents. Initial research in this area demonstrated that for two sites under analysis (the 1994 California election server and a corporate site), if connections were held open for 60 seconds, then 50% of the visitors to the site would receive at least 10 files per open connection. On average, each connection supported six requests, most connections were reused, and yet the number of open connections remained low [9]. Later analysis by the World Wide Web Consortium [10] showed that the current practice of maintaining parallel open connections (e.g., four connections for the Netscape browser) was crucial for achieving acceptable latency. If we apply these results to the persistent connection issue, then it's possible that it is necessary to maintain multiple persistent connections per session. We wanted to investigate the resource utilization effects due to maintaining multiple persistent connections per session.

Using a simulation based on a subset of our server logs, we explored four persistent connection parameters: the time-out interval, the maximum number of connections allowed per user, the maximum number of open persistent connections, and the algorithm for implementing dynamic time-out. Our simulator has three characteristics that detract from the behavior a server would observe in reality. First, we assume that each IP address corresponds to a single user, and therefore, can create only one session with a server. Second, the logs are biased toward browsers that make four concurrent connections, the standard Netscape browser behavior. Such a bias makes it impossible to accurately predict the user-perceived latency that will result when considering only one or two simultaneous connections. Third, in one of our logs (Harvard FAS),

Name of site	Date simulated	Time period	Number requests
Traditional Business	2/28/97	24 hours	11,549
Harvard FAS	2/28/96	4 hours	16,741
Free software	2/28/97	1 hour	26,574

Table 6: Persistent Connection Simulation Data.

the server does not record the latency between receiving a request and sending a response. In these cases, the simulation assumes 0-request-handling latency. Therefore, the FAS results will tend to be pessimistic about the effectiveness of persistent connections. The pessimism occurs when we assume data has been transmitted instantaneously and use that time as the "last active" time of the connection. In reality, the latest activity on that connection will occur after the response

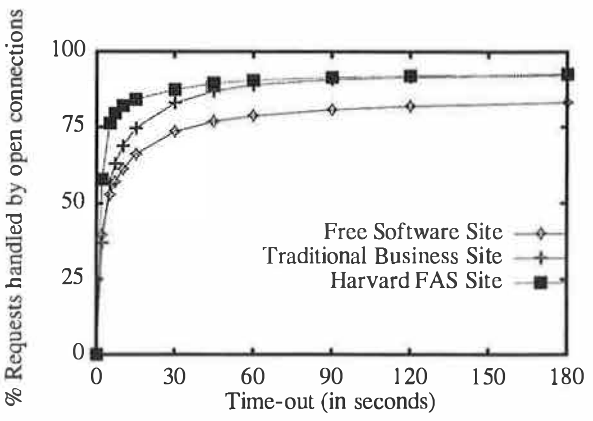


Figure 4: Sensitivity to Time-Out Intervals. In this simulation, we impose no maximum time-out limit, recording what percentage of requests could be handled by already open connections, as a function of the time-out interval. There is little additional benefit derived from leaving connections open longer than 15 seconds.

has been handled, and the possibility exists that we will time the connection out prematurely. Similarly, the estimations on the number of concurrently open connections for this site will tend to be small. In contrast, when we do have these latencies, then the simulation's time-out mechanism behaves exactly as a server's time-out mechanism. That is, the server begins the time-out period calculation as soon as it sends data over the connection, even though the client may receive the data much later, so the server's perception of how long a connection is idle may be significantly different from the perception of the client. While potentially suboptimal, this is the only knowledge that the server has, so it is used in timing out connections.

Table 6 describes the logs used, the dates and time periods that were run, and the number of requests processed. Although we chose our most heavily accessed sites, the time periods for each site differ because the levels of traffic vary so greatly. For each site, we selected four sets of each time period; we present the results of a single time period, but the results presented here are indicative for all the time periods.

We first ran the simulator with an infinite time-out interval, so we could determine the maximum benefit of persistent connections. In this simulation, the clients used only one persistent connection, generating the highest degree of connection reuse.

Figure 4 shows the persistent connection utilization as a function of the time-out interval. While the percentage of requests handled by persistent connections climbs rapidly up to a 15-second time-out, it remains relatively stable for intervals longer than 15 seconds. The greatest benefit for persistent connections

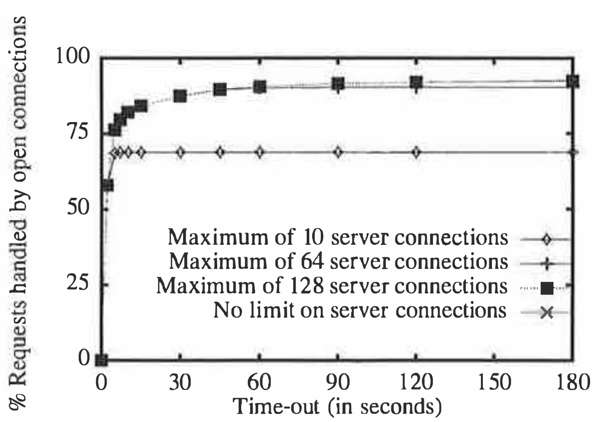


Figure 5: Sensitivity to Limiting the Number of Open Connections. This data shows the Free Software Site and the percentage of requests handled by persistent connections as we limit the maximum number of open connections. The connection limit can force connections to close prematurely reducing the benefit of the persistent connections.

occurs when users download a few pages, and these pages all use the same connection. On the FSS site, the majority of users exhibit this access pattern, so longer time-outs produce little benefit. Although users on the BUS site download more documents, they also derive little benefit from increased time-out intervals. Leaving connections open for longer intervals leads to a linear increase in the number of open connections. In fact, during the simulation of the site with the heaviest traffic (FSS), a three minute time-out resulted in over 300 open connections.

A more realistic analysis of persistent connections requires setting a limit on the number of open connections. Mogul focused his analysis on 10, 64, 128, and 1024 open connections. Our data shows that none of the servers opened as many as 1024 connections. In fact, only FAS and FSS ever had as many as 64 concurrently open connections. For the remainder of this discussion, we focus on FSS, because its logs include latencies and its heavier traffic enables a better analysis of the stresses that could significantly affect persistent connections.

Not surprisingly, the data show that, regardless of the connection limit and level of traffic, closing the least recently used connection leads to the best performance. Similarly unsurprising, whenever the time-out length leads to more active connections than permissible, increasing the time-out interval provides no improvement, because leaving the connections open longer exacerbates the situation, causing connections to be closed due to the imposed resource constraint, see Figure 5. Dynamic time-outs introduce no discernible benefit, because they effectively implement shorter static time-outs. The key insight is that the time-out interval and maximum open connections must be well-balanced. If fewer open connections are allowed than are necessary for the time-out interval, then connections will be closed prematurely. If more connections are allowed than the time-out interval warrants, the connections will be underutilized, wasting resource.

The second question we examined was how many persistent connections should be allowed per client. The HTTP/1.1 standard allows for up to two persistent connections per client [5], but we observed better resource utilization when clients are limited to a single persistent connection. The results presented in Figure 4 and Figure 5 were for a single persistent connection per client. Figure 6 shows what happens as we allow clients to have multiple persistent connections. The interaction of the number of persistent connections per client and the maximum number of open connections on the server results in worse resource utilization than might have been expected. Since the site is heavily loaded, allowing two connections per user doubles the number of open connections on the server. Therefore, at limits of 10 and 64 connections, the server closes connections more quickly than in the original model. As discussed before, this behavior has adverse effects on resource utilization. When we compare the resource utilization of the server allowing one persistent connection per user and 64 total connections to that of the server with two persistent connections per user and 128 total connections in Figure 6, we see the mild difference that we expect. Of course, such decisions do not come without cost. ISPs charge customers for extra connections; the implications of requiring the server to retain twice as many open connections have serious ramifications for the cost structure of service provision. Allowing two connections per client requires that servers potentially double the number of simultaneous open connections to achieve the high connection re-use rates we see in the one connection case. Unfortunately, at this point, we do not have enough data to incorporate the effect that more connections have on user response time.

6. Why Clients Wait

Long delays on the Web are often attributed to "overloaded servers," and researchers have cited four causes of server latency: the number of TIME_WAIT network connections, the number of concurrently active requests, the cost of CGI, and the sizes of the files requested [3][8]. Server logs provide rather incomplete latency measurements, but we can use the information available to determine that users accessing the servers we analyzed do experience long latencies that cannot be

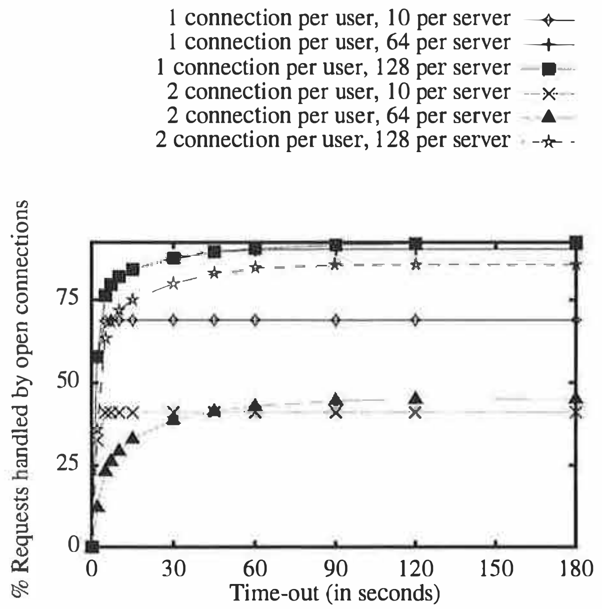


Figure 6: The Impact of Multiple Persistent Connections per Client. Allowing a client to create two persistent connections leads to the premature closing of many connections and a degradation in performance.

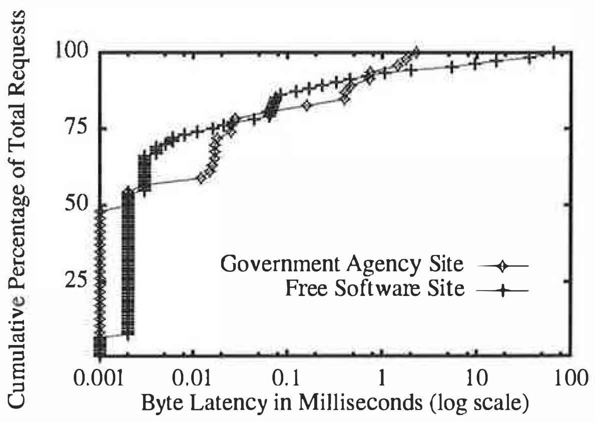


Figure 7: Byte Latencies for the Free Software Site (6722 hits/15 minutes) and the Government Agency Site (368 hits/15 minutes). Despite handling nearly 20 times as many requests as the Government Agency Site, the Software site shows similar byte-latencies.

attributed to the server. The latency logged by the server includes the time between the server initially receiving a request and the server issuing the last 8 KB write in response to the request. In particular, this time does not include connection setup (which happens before the server gets the request), the time to transmit the last block of data, or the effect of virtual hosting (supporting multiple Web sites on a single machine). Nonetheless, given the albeit limited data in server logs, we are still able to determine that, even for our most heavily accessed site (FSS), the server was not responsible for any user-perceptible latency.

For this analysis, we chose 15 minute segments of near-peak activity on three of the servers, representing three different orders of magnitude of traffic. For the purpose of this discussion, we will focus on the most heavily used server (FSS). During the peak interval, the server handled 6722 requests which equates to a server handling approximately 650,000 requests per day. This site is the most heavily used site hosted by our first ISP, which is one of the largest ISPs in the country.

The server for this site breaks requests into 8 KB chunks, waiting until each chunk has been acknowledged before sending the next one. On the last chunk, the server considers its job done as soon as it writes the data into its network buffers. Our first step was to analyze all requests smaller than 8 KB, in which case, the latency recorded by the server is exactly the time the server spent handling the request. Even during periods of heavy activity, all such requests were handled in less than one second, and 50% of the requests were handled in less than 1 ms. So, for small files, the server is not introducing the latencies that plague users of the Web.

For files larger than 8 KB, the latencies reported include the server overhead in addition to network and client delays. For our busiest site (FSS), these latencies range from 50 ms to 10 minutes; this is the time that users actually wait. We cannot directly compare these latencies because documents vary significantly in size, and we expect that it will take markedly longer to transfer a one megabyte file than a one kilobyte file. In order to analyze these requests accurately, we use the metric *byte-latency*, which is the average time that it takes the client to receive a single byte of data. Interestingly enough, when we compare byte-latencies across sites with loads that differ by more than an order of magnitude (FSS and GOV), Figure 7 shows that the distribution of byte-latencies is nearly identical and varies over four orders of magnitude; this kind of variation cannot be explained by any of the commonly proposed theories of server latency.

One common perception is that load and the number of open connections cause excessive server latencies. We assume that the number of active requests corresponds to server load, and examined the relationship between latency and both the average and maximum number of concurrent requests serviced while a request was being handled. Neither the average nor maximum shows any correlation to the byte-latencies.

Next we turn to the perception that CGI traffic is a cause of significantly increased latencies. However, during our intervals of peak activity, none of the CGI requests generated a response larger than 8 KB, and as mentioned earlier, none of the requests smaller than 8 KB required excessive processing time on the server.

Finally, we looked for a correlation between the size of the transfer and the byte-latency induced. Once again, there was no correlation.

From this series of analyses, we conclude that while some clients did observe long latencies from these servers, the latencies cannot be explained by server over-loading. The server has no difficulty handling most requests in under one ms, and the data from the server logs shows no indication that load, CGI, or file size contribute to the unpleasant latencies that users experience. We do find that the byte-latencies remain relatively fixed for given clients over 5, 10, and 15 minute intervals leading us to suspect that the bottleneck lies in the network, but we have no conclusive data to support this.

7. Conclusions

There seems to be common agreement that Web growth is exponential, but there has been no quantitative data indicating the magnitude of the exponent, nor the factors that cause this growth. Through server log analysis of a variety of sites, we have determined that site growth (in terms of number of hits) correlates with one of six different quantities: the number of Web users, the number of documents a user is likely to visit on a site, the number of documents on a site, the fee structure for accessing data, the frequency with which search engines return a particular site, and the efforts of Web masters at attracting users. In addition, we have dispelled certain widely held perceptions: that CGI is becoming increasingly important in general and that heavily loaded servers are the main cause of Web latency. Finally, we quantified the effects that key design parameters have in maximizing the resource utilization of persistent connections. There remains much work to be done. In particular, detailed analysis of some of the most heavily accessed sites on the Web would be generally useful to the research community. And, while we have ruled out certain causes for latency, the answer to the question, "Why do users wait on the Web?" still eludes the research community.

8. Bibliography

[1] Bestavros, A., "WWW Traffic Reduction and Load Balancing Through Server-Based Caching," *IEEE Concurrency: Special Issue on Parallel and Distributed Technology*, vol. 5, pp. 56-67, Jan-Mar 1997.

[2] Bowman, C., Danzig, P., Hardy, D., Manber, U., Schwartz, M., The Harvest Information Discovery and Access System. Computer Networks and ISDN Systems 28 (1995) pp. 119-125.

[3] Edwards, N., Rees, O. "Performance of HTTP and CGI," Available at `http://www.ansa.co.uk/ANSA/ISF/1506/APM1506.html`

[4] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T., Hypertext Transfer Protocol—HTTP/1.1. *Internet Engineering Task Force Working Draft*, August 1996.

[5] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T., Hypertext Transfer Protocol—HTTP/1.1, RFC-2068, ftp://ds.internic.net/rfc/rfc2068.txt.

[6] Gwertzman, J., Seltzer, M., "The Case for Geographical Push-Caching," Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, Orcas Island, WA, May, 1995, 51–55.

[7] Manley, S., "An Analysis of Issues Facing World Wide Web Servers," Harvard University, Computer Research Laboratory Technical Report, TR-12-97, July 1997.

[8] Mogul, J., "Network Behavior of a Busy Web Server and its Clients," Digital Equipment Corporation Western Research Lab Technical Report DEC WRL RR 95.5.

[9] Mogul, J., "The Case for Persistent Connection HTTP," *Proceedings of the 1995 SIGCOMM '95 Conference on Communications Architectures and Protocols.*

[10] Nielsen, H., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H., Lilley, C., Network Performance Effects of HTTP/1.1, CSS1, and PNG. W3 Consortium Note available at `http://www.w3.org/pub/WWW/Protocols/HTTP/Performance/Pipeline.html`

SPAND: Shared Passive Network Performance Discovery

Srinivasan Seshan
srini@watson.ibm.com

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Mark Stemm, Randy H. Katz
{stemm,randy}@cs.berkeley.edu

Computer Science Division
University of California at Berkeley
Berkeley, CA 94720

Abstract

In the Internet today, users and applications must often make decisions based on the performance they expect to receive from other Internet hosts. For example, users can often view many Web pages in low-bandwidth or high-bandwidth versions, while other pages present users with long lists of mirror sites to chose from. Current techniques to perform these decisions are often ad hoc or poorly designed. The most common solution used today is to require the user to manually make decisions based on their own experience and whatever information is provided by the application. Previous efforts to automate this decision-making process have relied on *isolated, active* network probes from a host. Unfortunately, this method of making measurements has several problems. Active probing introduces unnecessary network traffic that can quickly become a significant part of the total traffic handled by busy Web servers. Probing from a single host results in less accurate information and more redundant network probes than a system that shares information with nearby hosts. In this paper, we propose a system called SPAND (**S**hared **Pa**ssive **N**etwork Performance **D**iscovery) that determines network characteristics by making *shared, passive* measurements from a collection of hosts. In this paper, we show why using passive measurements from a collection of hosts has advantages over using active measurements from a single host. We also show that sharing measurements can significantly increase the accuracy and timeliness of predictions. In addition, we present a initial prototype design of SPAND, the current implementation status of our system, and initial performance results that show the potential benefits of SPAND.

1. Introduction

In today's Internet, it is impossible to determine in advance what the network performance (e.g., available bandwidth, latency, and packet loss probability) between a pair of Internet hosts will be. This capability is missing in today's suite of Internet services, and many applications could benefit from such a service:

- Applications that are presented with a choice of hosts that replicate the same service. Specific examples of this are FTP and Web mirror sites and Harvest caches that contact the "closest" peer cache. Today, these applications rely on statistics such as hop count/routing metrics [9], round-trip latency [7], or geographic location [10]. However, each of these techniques has significant weaknesses, as we show in Section 2.
- Web clients that have a choice of content *fidelity* to download from a Web server, e.g., a full graphics representation for high-bandwidth connectivity or a text-only representation for low-bandwidth connectivity. Today, the user must manually select the fidelity of the content that they wish to view, sometimes making overaggressive decisions such as viewing no images at all.
- Applications that provide feedback to the user that indicates the expected performance to a distant site. For example, Web browsers could insert an informative icon next to a hyperlink indicating the expected available bandwidth to the remote site referred to by the hyperlink.

Each of these applications needs the ability to determine in advance the expected network performance between a pair of Internet hosts. Previous work in this area has relied on *isolated, active* network probing from a single host to determine network performance characteristics. There are two major problems with this approach:

- Active probing requires the introduction of unnecessary traffic into the network. Clearly, an approach that determines the same information with a minimum of unnecessary traffic is more desirable. We also show later that this unnecessary traffic can quickly grow to become a non-negligible part of the traffic reaching busy Web servers, reducing their efficiency and sometimes their scalability.

- Probing from a single host prevents a client from using the past information of nearby clients to predict future performance. Recent studies [2][20] have shown that network performance from a client to a server is often stable for many minutes and very similar to the performance observed by other nearby clients, so there are potential benefits of sharing information between hosts. In Section 3.2, we show examples where using shared rather than isolated information increases the likelihood that previously collected network characteristics are valid.

We are developing a system called SPAND (**S**hared **Pas**sive **N**etwork Performance **D**iscovery) that overcomes the above problems of isolated active probing by collecting network performance information *passively* from a collection of hosts, *caching* it for some time and *sharing* this information between them. This allows a group of hosts to obtain timely and accurate network performance information in a manner that does not introduce unnecessary network traffic.

The rest of this paper is organized as follows. In Section 2, we describe related work in more detail. In Section 3, we point out the advantages and challenges of using passive shared measurements over isolated active measurements. In Section 4, we present a detailed design of SPAND. In Section 5, we describe the implementation status of SPAND and initial performance results, and in Section 6, we conclude and describe future work.

2. Related Work

In this section, we describe in more detail previous work in network probing algorithms and server selection systems.

2.1 Probing Algorithms

A common technique to estimate expected performance is to test the network by introducing probe packets. The objective of these probes is to measure the round trip latency, peak bandwidth or available "fair-share" bandwidth along the path from one host to another

Probes to measure round-trip latency and peak bandwidth are typically done by sending groups of back-to-back packets to a server which echoes them back to the sender. These probes are referred to as NetDyn probes in [4], packet pair in [13], and bprobes in [6]. As pointed out in earlier work on TCP dynamics [12], the spacing between these packets at the bottleneck link is preserved on higher-bandwidth links and can be measured at the sender.

If the routers in the network do not implement fair queuing, the minimum of many such measurements is likely to be close to the raw link bandwidth, as assumed in other work ([4][6][19]). Pathchar [19] combines this technique with traceroute [22] to measure the link bandwidth and latency of each hop along the path from one Internet host to another. Packet Bunch Mode (PBM) [20] extends this technique by analyzing various sized groups of packets inserted into the network back-to-back. This allows PBM to handle multi-channel links (i.e. ISDN connections, muiti-link Point-to-Point Protocol (PPP) links, etc.) as well as improve the accuracy of the resulting measurements.

If routers in the network implement fair queuing, then the bandwidth indicated by the back-to-back packet probes is an accurate estimate of the "fair share" of the bottleneck link's bandwidth [13]. Another fair share bandwidth estimator, cprobe [6], sends a short sequence of echo packets from one host to another as a simulated connection (without minimal flow control and no congestion control). By assuming that "almost-fair" queuing occurs over the short sequence of packets, cprobe provides an estimate for the available bandwidth along the path from one host to another. Combined with information from bprobes, cprobes can estimate the competing traffic along the bottleneck link. However, it is unclear how often this "almost-fair" assumption is correct and how accurate the resulting measurements are. TReno [15] also uses ICMP echo packets as a simulated connection, but uses flow control and congestion control algorithms equivalent to that used by TCP.

The problem with these methods is that they can introduce significant amounts of traffic that is not useful to any application. For example, pathchar sends at least tens of kilobytes of probe traffic per hop, and a cprobe sends 6 kilobytes of traffic per probe. This amount of probe traffic is a significant fraction (approximately 20%) of the mean transfer size for many Web connections ([1], [2]) as well as a significant fraction of the mean transfer size for many Web sessions. We discuss in more detail the limitations of active probing in Section 3.3.

2.2 Server Selection Systems

Many server selection systems use network probing algorithms to identify the closest or best connected server. For example, Carter et al. at Boston University [5] use cprobes and bprobes to classify the connectivity of a group of candidate mirror sites. Harvest [7] uses round-trip latency to identify the best peer cache from which to retrieve a Web page. Requests are initiated to

System	What measured/ used to identify performance	Additional traffic introduced	Notes	Where Deployed
Bprobes, Cprobes	Peak and Available Bottleneck Bandwidth	Significant (~10K)	Cprobes uses no flow/ congestion control	Client Side
Packet Pair	Available Bandwidth	Little (~1K)	Assumes per-flow fair queuing	Client Side
Pathchar	Hop-by-hop link bandwidth, latency	Significant (>10K)	No congestion control	Client Side
Packet Bunch Modes	Peak Bottleneck Bandwidth	Significant (~10K)		Client Side
Treno	Available Bandwidth	Significant (>10K)	Uses TCP Flow/Congestion Control	Client Side
IPV6 Anycast	Routing Metric	Little (routing data and queries)		Internal Network
Harvest	Latency	Little (~1K)		Server Side
HOPS	Routing Metric	Little (routing data and queries)		Internal Network
DistributedDirector	Routing Metric	Little (routing data and remote queries)		Server Side and Internal Network
SPAND	**Available Bandwidth, Packet Loss Probability**	**Little (local reports and queries)**		**Client Side**

TABLE 1. Summary of Previous Work compared to SPAND

each peer cache, and the first to begin responding with a positive answer is selected and the other connections are closed. Other proposals [10] rely on geographic location for selecting the best cache location when push-caching Web documents.

There are also preliminary designs for network-based services to aid in server selection. IPV6's Anycast [11][18] service provides a mechanism that directs a client's packets to any one of a number of hosts that represent the same IP addresses. This service uses routing metrics as the criteria for server selection. The Host Proximity Service (HOPS) [9] uses routing metrics such as hop counts to select the closest one of a number of candidate mirror sites.

Cisco's DistributedDirector [8] product relies on measurements from Director Response Protocol (DRP) servers to perform efficient wide area server selection. The DRP servers collect Border Gateway Protocol (BGP) and Interior Gateway Protocol (IGP) routing table metrics between distributed servers and clients. When a client connects to a server, DistributedDirector contacts the DRP server for each replica site to retrieve the information about the distance between the replica site and the client.

The problem with many of these approaches is that one-way latency, geographic location, and hop count are often poor estimates of actual completion time. Other work [5][17] shows that hop count is poorly correlated with available bandwidth, and one-way latency does not take available bandwidth into account at all. Even those systems that provide better performance metrics [5] rely on each end host independently measuring network performance.

Another design choice to consider is where the system must be deployed. A system that is deployed only at the endpoints of the network is easier to maintain and deploy than a system that must be deployed inside the internal infrastructure, and a system that is deployed only at the client side is easier to deploy than a system that relies on client and server side components.

Table 1 summarizes the previous work in this area. The significant shortcomings of existing network performance discovery and server selection systems are:

- Introduction of new traffic into the network that can quickly become significant when compared to "useful" traffic.
- Reliance on measurements from a single host, which are more often redundant and inaccurate than measurements from a collection of hosts.
- Use of metrics such as hop count, latency, and geographic location as imprecise estimates of available bandwidth.

We discuss these shortcomings further in the next section.

3. Passive and Cooperative Measurements

The goal of our work is to provide a unified repository of actual end-to-end performance information for applications to consult when they wish to determine the network performance to distant Internet hosts. Our approach addresses the shortcomings of previous work in 2 ways: (1) relying solely on passive rather than active measurements of the network, and (2) sharing measurement information between all hosts in a local network region. In this section we show the potential benefits and challenges of using shared, passive measurements to predict network performance instead of using isolated, active measurements.

3.1 Network Performance Stability

In order for past transfers observed by hosts in a region to accurately predict future performance, the network performance between hosts must remain relatively stable for periods of time on the order of minutes. Without this predictability, it would be impossible for shared passive measurements of the network to be meaningful. Past work has shown evidence that this is true for some Internet hosts [2][20]. We wanted to verify these results in a different scenario.

To understand more closely the dynamics of network characteristics to a distant host, we performed a controlled set of network measurements. This consisted of repeated HTTP transfers between UC Berkeley and IBM Watson. For a 5 hour daytime period (from 9am PDT to 2pm PDT), a Web client at UC Berkeley periodically downloaded an image object from a Web server running at IBM Watson. Although this measurement is clearly not representative of the variety of connectivity and access patterns that exist between Internet hosts, it allowed us to focus on the short-term changes in network characteristics that could occur between a pair of well-connected Internet hosts separated by a large number of Internet hops.

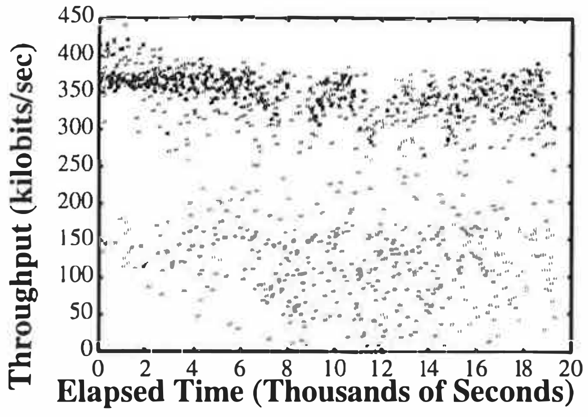


Figure 1. Throughput from UC Berkeley to IBM during a 5 hour daytime period

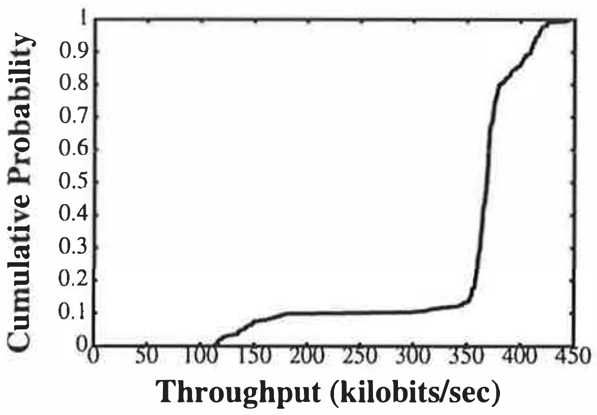


Figure 2. CDF of throughput from UC Berkeley to IBM: initial 30 minutes

Figure 1 shows the raw throughput measurements as a function of time over the 5 hour period. We see that in the first 30 minutes of the trace, one group of measurements is clustered around 350 kilobits/sec (presumably the available bandwidth on the path between UC Berkeley and IBM). A smaller group of measurements has lower throughputs, at 200 kilobits/sec and lower. This second group of connections presumably experiences one or more packet losses. This clustering is more clearly shown in Figure 2, the cumulative distribution function (CDF) of throughputs during the first 30 minutes of the trace. As the day progresses, two things change. The available bandwidth decreases as the day progresses, and a larger fraction of transfers experience one or more packet losses. This effect is shown in Figure 3, the cumulative distribution function of throughputs for the entire 5 hour period. More samples are clustered around lower throughput values and there is more variation in the available bandwidth. However, there is still a noticeable separation between the two groups of throughput measurements.

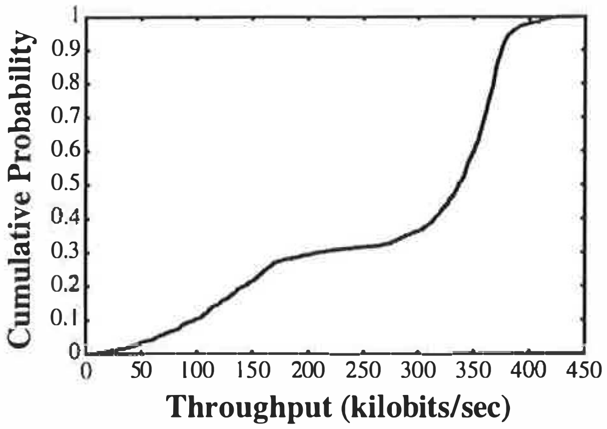


Figure 3. CDF of throughput from UC Berkeley to IBM: full 5 hour period

This distribution of performance suggests that although the distribution of throughputs changes as the day progresses, a system like SPAND could still provide meaningful performance predictions. Although the performance distribution of the early part of the day and the later part of the day are quite different, they each have lifetimes of tens of minutes or more. Even when aggregating all the different performance measurements for the entire 5 hour period, approximately 65% of the throughput samples are within a factor of 2 of the median throughput.

3.2 Shared Measurements: Benefits and Challenges

Using shared rather than isolated measurements allows us to eliminate redundant network probes. If two hosts are nearby each other in terms of network topology, it is likely that they share the same bottleneck link to a remote site. As a result, the available bandwidth they measure to this site is likely to be very similar [2]. Therefore, it is unnecessary and undesirable for each of these hosts to independently probe to find this information--they can learn from each other.

To quantify how often this information can be shared between nearby hosts, we examined Internet usage patterns by analyzing client-side Web traces. From these traces, we determined how often a client would need to probe the network to determine the network performance to a particular Web server when shared information was and was not available.

More formally, for a single Web server, we can represent the list of arrival times from a single client (or a shared collection of clients) as a sequence (t_1, t_2,..., t_n). If the difference between t_{i+1} and t_i is extremely small (less than ten seconds), we merge the events together into a single Web browsing "session." Clearly, the first arrival always requires a probe of the network. In addition, if

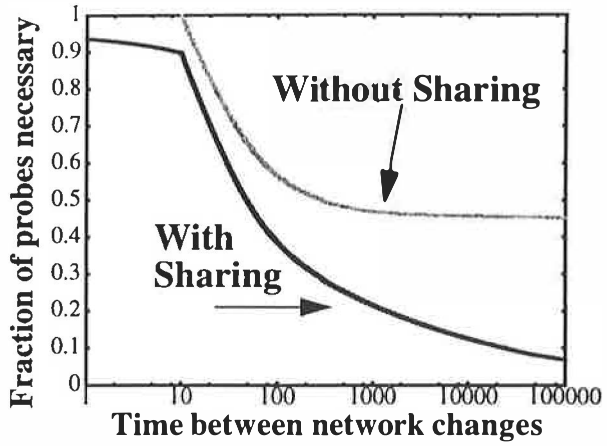


Figure 5. The benefit of sharing. Figure shows the fraction of network probes that are necessary as a function of the time between network state changes

we assume that the time between significant network changes is a fixed value Δ, then if t_{i+1}-t_i>Δ, then the client must make a probe to determine the new network characteristics. If t_{i+1}-t_i<Δ, then no probe is necessary. As mentioned previously [2][20], an appropriate value for Δ is on the order of tens of minutes.

Figure 5 shows the results of this analysis for a particular client-side trace consisting of 404780 connections from approximately 600 users over an 80 hour time period [23]. The x-axis represents the time Δ between network changes, and the y-axis represents the fraction of network probes that are necessary. There are two curves in the figure. The upper curve represents the number of probes that are necessary if no sharing between clients is performed, and the lower curve represents the number of probes that are necessary if clients share information between them. The upper curve begins at Δ=10 seconds because of the "sessionizing" of individual connections described above. We see that the number of probes that are necessary when clients share network information is dramatically reduced. This is evidence that a collection of hosts can eliminate many redundant network probes by sharing information.

The use of shared measurements is not without challenges, however. Measurements from arbitrary hosts in a region cannot be combined. For example, it is necessary to separate modem users within a local domain from LAN users in the same domain, because the two sets of users may not share the same bottleneck link. Similarly, hosts in a local domain may use different TCP implementations that result in widely varying performance. The challenge is that it is often difficult to determine who the set of "similarly connected" hosts within an local domain are. We can use the topology of the local domain along with post-processing on past measure-

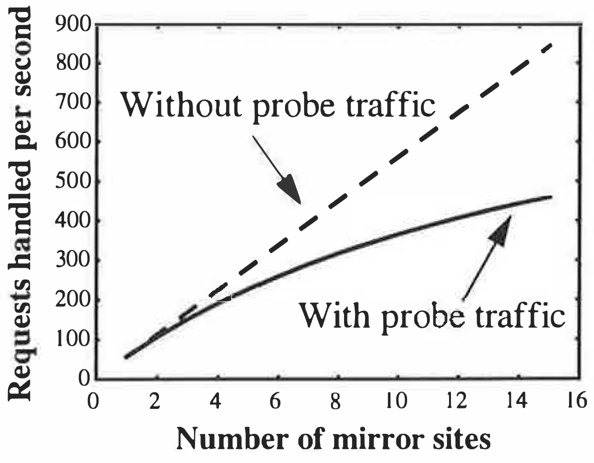


Figure 6. The effect of probe traffic on scalability. Figure shows requests/second that mirrors can serve as a function of the number of mirror sites

ments to determine which network subnets exhibit significantly different performance. The system can then coalesce these subnets together into classes of equivalent connectivity and avoid aggregating measurements from dissimilar hosts.

3.3 Passive Measurements: Benefits and Challenges

The use of passive measurements avoids the introduction of useless probe traffic into the network. This advantage over active probing systems comes at the expense of making the job of measuring available bandwidth more difficult. However, this advantage is critical since probe traffic can sometimes become a measurable fraction of the traffic handled by busy Web servers. For example, consider the scenario of mirror sites that replicate the same content. In an active probing system, a client must first contact each of the mirror sites to determine which mirror is the "best." This slows down servers with probe-only traffic and limits the scalability of such a system.

The following thought experiment shows why. Consider a Web server with a variable number of mirror sites. Assume that each mirror site is connected to the Internet via a 45 Mbit/second T3 link and assume that the mean transfer size is 100 kbytes and the mean probe size is 6 kbytes. These are optimistic estimates; most Web transfers are shorter than 100 kbytes and many of the network probing algorithms discussed in Section 2.1 introduce more than 6 kbytes. From a network perspective, an estimate of the number of requests per second that the collection of mirrors can support is the aggregate bandwidth of the mirrors' Internet links divided by the sum of the average Web transfer size and any associated probe traffic for the transfer. Figure 6 shows the number of requests per second that such a system can support as a function of the number of mirror sites for two systems: one without probe traffic, and one with probe traffic. We see that the system without probe traffic scales perfectly with the number of mirrors. For the system with probe traffic, however, for each Web request that is handled by a single mirror, a network probe must be sent to all of the other mirrors. On the server side, this means that for each Web request a particular mirror site handles, it must also handle a probe request from clients being serviced at every other mirror location. As the number of mirrors increases, the number of requests served per second becomes limited by the additional probe traffic.

There are challenges in using passive network measurements, however. Using passive rather than active measurements is difficult for several reasons:

- Passive measurements are uncontrolled experiments, and it can be difficult to separate network events from those occurring at the endpoints, such as a rate-limited transmission or a slow or unreachable server.
- Passive measurements are only collected when a host contacts a remote site. In order to have timely measurements, hosts in a local domain must visit distant hosts often enough to obtain timely information. If this is not true, the client may obtain either out-of-date information or no information at all.

For our purposes, there is no need to distinguish between network events and endpoint events. If a remote site is unreachable or has poor connectivity because it is down or overloaded, that information is just as useful. It is important to distinguish between rate-controlled and bulk transfer connections so the performance numbers from one are not used to estimate performance for the other. We can distinguish between rate-controlled and bulk transfer transmissions by using TCP and UDP port numbers and the application classes described in Section 4.

To identify if passive measurements can provide timely information, we must analyze typical Internet usage patterns and determine how often passive techniques lead to out-of-date information. We can use the results shown in Figure 5 to see this. We can model the arrival pattern of clients as a sequence of times ($t_1 \ldots t_n$) as before. In the passive case, when $t_{i+1}-t_i>\Delta$, instead of saying that an active probe is necessary, we say that the passively collected information has become out of date. So the fraction of time that an active probe is necessary is exactly the same as the fraction of time that passive measurements become out of date. As mentioned earlier, the appropriate value for Δ is on the order of tens of min-

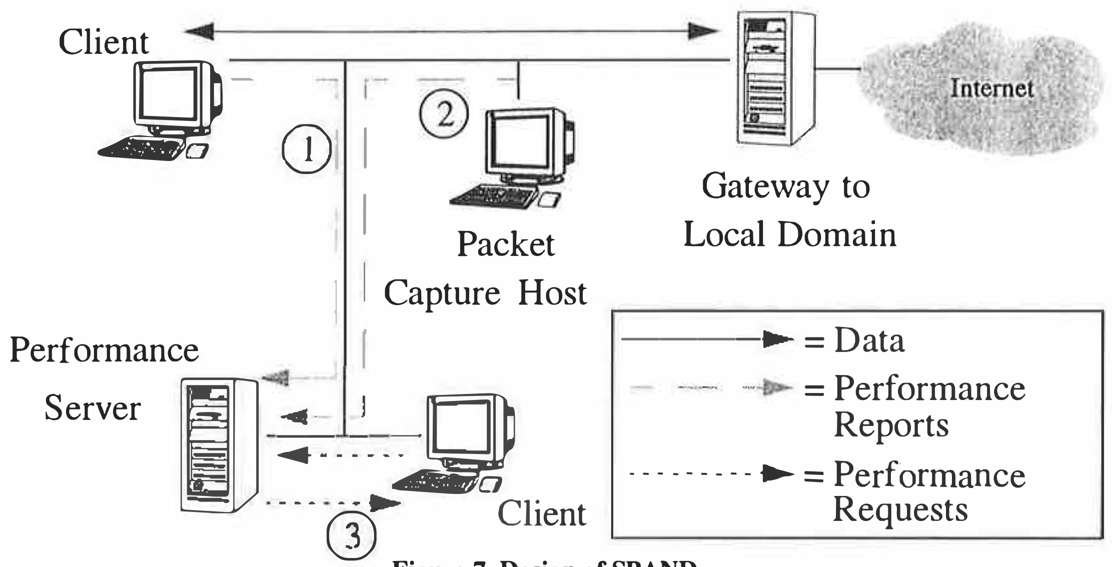


Figure 7. Design of SPAND

utes. We see that even a relatively small collection of hosts can obtain timely network information when sharing information between them. If we assume that network conditions change approximately every 15 minutes, then the passive measurements collected from this relatively small collection of 600 hosts will be accurate approximately 78% of the time. For larger collections of hosts (such as domain-wide passive measurements), the availability of timely information will be even greater, as shown in Section 5.3.

4. Design of the SPAND System

In this section, we describe the design for SPAND, including steps for incremental deployment in existing networks.

Figure 7 shows a diagram of the components of SPAND. SPAND is comprised of *Clients, Performance Servers, and Packet Capture Hosts.* Clients have modified network stacks that transmit *Performance Reports* to Performance Servers. These reports contain information about the performance of the network path between the client and distant hosts. The format of a Performance Report is shown in Figure 8, and includes parameters such as connection bandwidth and packet loss statistics. The Transport Protocol field indicates the type of transport connection (UDP or TCP) used by the initiator of the connection. The optional Application Class field is a hint as to the way in which the application is using the transport connection. If an Application Class is not provided, the Performance Server can use the Port Number and Transport Protocol fields to make a guess for the application class.

The Application Class field is desirable because not all

0	15	16	31
Version	**Type**	**Transport Pr.**	**App.Class**
Source IP Address			
Source Port		**Dest Port**	
Dest IP Address			
NTP Timestamp, most sig word			
NTP Timestamp, least sig word			
Length of Sample in octets			
Duration of Sample in ms			
Total Packets Received			
Total Packets Lost			
Packet Size in octets			

Figure 8. Format of a Performance Report

applications use transport connections in the same way. Some applications (such as Web browsers and FTP programs) use TCP connections for bulk transfers and depend on the reliability, flow control, and congestion control abstractions that TCP connections provide. Applications such as telnet primarily use TCP connections for reliability and not for flow or congestion control. Other applications such as RealAudio in TCP mode use TCP connections for different reasons such as the ability to traverse firewalls. The transport level network performance reported from different applications may vary widely depending on the way the transport connection is used, and we want to separate these performance reports into distinct classes.

0	15	16	31
Version	**Type**	**Protocol**	**App. Class**
Request IP Address			

Figure 9. Format of a Performance Request

0	15	16	31
Version	**Type**	**Protocol**	**App. Class**
Response IP Address			
Expected Available Bandwidth (kbits/sec)			
Std Dev of Available Bandwidth (kbits/sec)			
Expected Packet Loss Probability			
Std Dev of Expected Loss Probability			

Figure 10. Format of a Performance Response

In addition, many applications may intermittently change the way in which a connection is used. For example, a passive FTP connection may switch from transporting control information to transferring bulk data. Similarly, a persistent HTTP 1.1 connection may have idle "think" periods where the user is looking at a Web page as well as bulk transfer periods. To properly account for all these variations, we need applications to take part in the performance reporting process. Our toolkit provides an API which enables applications to start a measurement period on a connection as well as end the measurement and automatically send a report to the local Performance Server.

A Performance Server receives reports from all clients in a local area and must incorporate them into its performance estimates. The server maintains different estimates for different classes of applications as well as different classes of connectivity within its domain. In addition, the Performance Server can also identify reports that have possibly inaccurate information and discard them. Clients may later query the information in the server by sending it a Performance Request containing an (Address, Application Class) pair. The server responds to it by returning a Performance Response for that pair, if one exists. This response includes the Performance Server's estimates of available bandwidth and packet loss probability from the local domain to the specified foreign host. The request and response formats are shown in Figures 9 and 10.

4.1 Mechanisms for Incremental Deployment

The system described in the previous section is an ideal endpoint we would like to reach. In practice, it may be difficult to immediately modify all client applications to generate performance reports, especially since many clients may need modifications to their protocol stacks to make the statistics necessary for SPAND available. To quickly capture performance from a large number of end clients, a Packet Capture Host can be deployed that uses a tool such as BPF [16] to observe all transfers to and from the clients. The Packet Capture Host determines the network performance from its observations and sends reports to the Performance Server on behalf of the clients. This allows a large number of Performance Reports to be collected while end clients are slowly upgraded. The weakness of this approach is that a number of heuristics must be employed to recreate application-level information that is available at the end client. Section 5.2 describes these heuristics in more detail.

4.2 Example Scenario

This example scenario using Figure 7 illustrates the way in which the agents that make up SPAND coordinate. Assume that a user is browsing the Web. As the user is browsing, the user's application generates Performance Reports and sends them to the local Performance Server (1 in the figure). Also, a Packet Capture Host deployed at the gateway from the local domain to the Internet generates Performance Reports on behalf of the hosts in the domain (2 in the figure). Later, some other user reaches a page where she must select between mirror locations. The Web browser makes a Performance Query to the local Performance Server and gets a response (3 in the figure). The Web browser uses the Performance Response to automatically contact the mirror site with the best connectivity.

5. Implementation Status and Performance

In this section, we describe the current implementation status of SPAND and present initial performance measurements from a working SPAND prototype.

5.1 Implementation Details

SPAND is organized as a C++ toolkit that provides object abstractions for the agents described above. Application writers can create objects for agents such as PerformanceReporter(), PerformanceReportee(), PerformanceRequestor(), and PerformanceRequestee() and use these objects to make, send or receive reports, or make queries about network performance. We also have partial implementations of the toolkit in Java and Perl.

Using the SPAND toolkit, we have implemented the Packet Capture Host, Performance Server and a simple text-based SPAND Client. We have also written several client applications that use the SPAND toolkit to make use of Performance Reports. We have written a HTTP proxy using the Perl libwww [14] library and the SPAND toolkit that modifies HTML pages to include informative icons that indicate the network performance to a distant site mentioned in a hyperlink. This is not the first application of this type; others have been developed at IBM [3] and at Boston University [5]. However, our application uses actual observed network performance from local hosts to make decisions about the icon to insert in the HTML page.

We have also written a Java-based application that allows the user to obtain an overview of the connectivity from a local domain to distant hosts. This application shows the number of performance reports collected for all hosts as well as the details about the reported network statistics for a given host. This tool was used to generate the graphs in Section 3.1.

5.2 Packet Capture Host Policies

Because our Packet Capture Host is not located at end clients, it does not have perfect information about the way in which applications use TCP connections. This can lead to inaccurate measurements of network characteristics such as bandwidth. For example, if a Web browser uses persistent or keep-alive connections to make many HTTP requests over a single TCP connection, then simply measuring the observed bandwidth over the TCP connection will include the gaps between HTTP requests in the total time of the connection, leading to a reduction in reported bandwidth. To account for this effect, we modified the Packet Capture Host to use heuristics to detect these idle periods in connections. When the Packet Capture Host detects an idle period, it makes two reports: one for the part of the connection before the idle period, and another for the part of the connection after the idle period. The ratio measurements in Section 5.3 include systems with and without the use of these heuristics. Another example is when Common Gateway Interface (CGI) programs are executed as part of HTTP requests. The idle time when the server is executing the CGI program leads to an artificially low reported bandwidth and does not reflect the performance of other HTTP requests to the same server. Ideally, the Packet Capture Host would treat these connections as a different Application class. For the purposes of these measurements, however, the Packet Capture Host excluded the idle periods and generated multiple Performance Reports as above.

5.3 Performance

There are several important metrics by which we can measure the performance of the SPAND system:

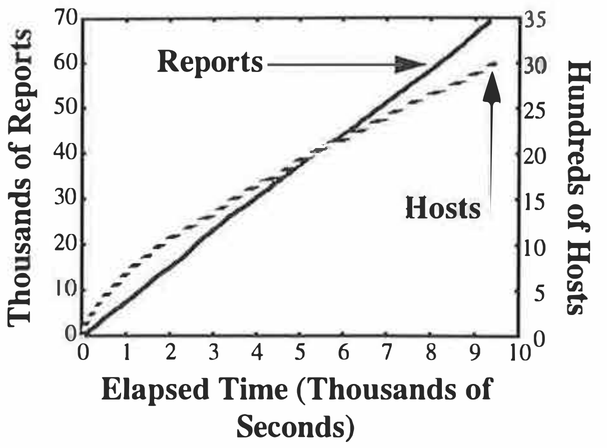


Figure 11. Cumulative number of reports generated and hosts reported about as a function of time

1. How long does it take before the system can service the bulk of Performance Requests?
2. In the steady-state, what percentage of Performance Requests does the system service?
3. How accurate are the performance predictions?

To test the performance of our system, we deployed a Packet Capture Host at the connection between IBM Research and its Internet service provider. Hosts within IBM communicate with the Internet through the use of a SOCKSv4 firewall [21]. This firewall forces all internal hosts to use TCP and to initiate transfers (i.e. servers can not be inside the firewall). The packet capture host monitored all traffic between the SOCKS firewall at IBM Research and servers outside IBM's internal domain. The measurements we present here are from a 3 hour long weekday period. During this period, 62,781 performance reports were generated by the packet capture host for 3,008 external hosts. At the end of this period, the Performance Server maintained a database of approximately 60 megabytes of Performance Reports. Figure 11 shows the cumulative number of reports generated and hosts reported about as a function of time.We see that about 10 reports are generated per second, which results in a network overhead of approximately 5 kilobits per second. We also see that while initially a large number of reports are about a relatively small number of hosts (the upward curve and leveling off of the curve), as time progresses, a significant number of new hosts are reported about as time progresses. This indicates that the "working set" of hosts includes a set of hosts who are reported about a small number of times. This finding is reinforced in Figure 12, which shows a histogram of the number of reports received for each host over the trace. We see that a large majority of hosts receive only a few

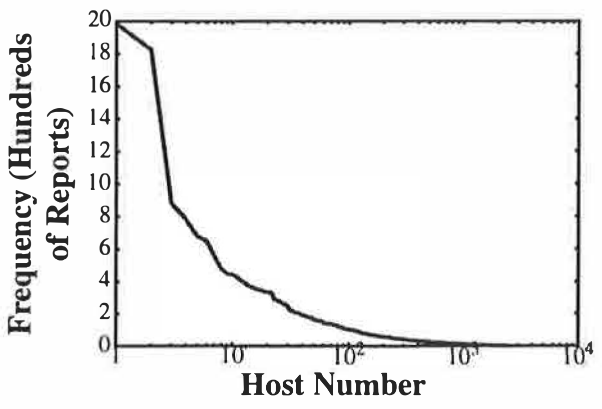


Figure 12. Histogram of number of Performance Reports received per host. The x axis is on a log scale.

reports, while a small fraction of hosts receive most of the reports. The mean number of reports received per host was 23.67 and the median number of reports received per host was 7.

To test the accuracy of the system, we had to generate a sequence of Performance Reports and Performance Requests to test the system. Since there are no applications running at IBM that currently use the SPAND system, we assumed that each client host would make a single Performance Request to the Performance Server for a distant host before connecting to that host, and a single Performance Report to the Performance Server after completing a connection. In actual practice, applications using SPAND would probably request the performance for many hosts and then make a connection to only one of them. The performance of the SPAND system on this workload is summarized in Figures 13 and 14.

When a Performance Server is first started, it has no information about prior network performance and cannot respond to many of the requests made to it. As the server begins to receive performance reports, it is able to respond to a greater percentage of requests. Determining the exact "warmup" time before the Performance Server can service most requests is important. Figure 13 shows the probability that a Performance Request can be serviced by the Performance Server as a function of the number of reports since the "cold start" time. We say that a request can be serviced if there is at least one previously collected Performance Report for that host in the Performance Server's database. As we can see from the graph, the Performance Server is able to service 70% of the requests within the first 300 reports (less than 1 minute), and the Performance Server reaches a steady-state service rate of 95% at around 10,000 reports (approximately 20 minutes). This indicates that when a

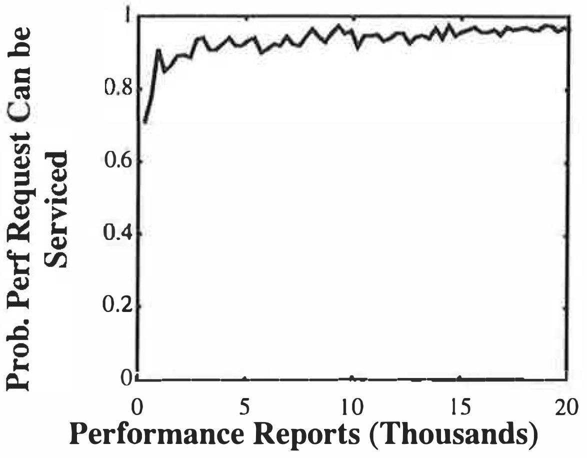


Figure 13. Probability that a Performance Request can be Serviced as a function of the number of Performance Reports.

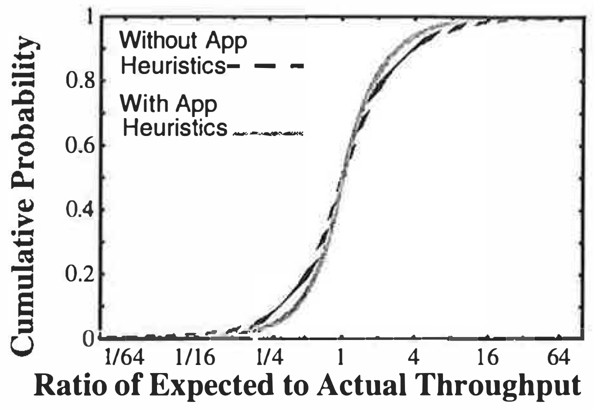


Figure 14. CDF of ratio of expected throughput (as generated by the Performance Server) to actual throughput (as reported by the client). The x axis is on a log scale

Performance Server is first brought up, there is enough locality in client access patterns that it can quickly service the bulk of the Performance Requests sent to it.

To measure the accuracy of Performance Responses, for each connection we computed the ratio of the throughput returned by the Performance Server for that connection's host with the throughput actually reported by the Packet Capture Host for that connection. Figure 14 plots the cumulative distribution function of these ratios. The x axis is plotted on a log scale to equally show ratios that are less than and greater than one. Table 2 shows the probability that a Performance Response is within a factor of 2 and 4 of the actual observed throughput. We see that Performance Responses are often close to the actual observed throughput. Obviously, different applications will have different requirements as to the error that they can tolerate. Factors of 2 and 4 are shown only as repre-

System	% within factor of 2	% within factor of 4
Base System	59.08%	84.05%
Base System + App Heuristics	68.84%	90.18%

TABLE 2. Accuracy of Performance Responses

sentative data points.

Closer examination of transfers that result in very large or small ratios indicate a few important sources of error in the predictions. Some servers have a mixture of small and large transfers made to them. As a result of TCP protocol behavior, the larger transfers tend to be limited by the available bandwidth to the server whereas the smaller transfers tend to limited by the round trip time to the server. We plan to incorporate round trip measurements into a future version of the SPAND Performance Server to address this problem. In addition, the heuristics used to recreate end client information are effective but not always accurate. Many poor estimates are a result of pauses in the application protocol. This source of inaccuracy will vanish as more end clients participate in the SPAND system and less reliance is made on the Packet Capture Host's heuristics.

6. Conclusions and Future Work

There are many classes of Internet applications that need the ability to predict in advance the network performance between a pair of Internet hosts. Previous work providing this information has depended on isolated, active measurements from a single host. This does not scale to many users and does not provide the most accurate and timely information possible. In this paper, we have proposed a system called SPAND (Shared Passive Network Performance Discovery) that uses passive measurements from a collection of hosts to determine wide-area network characteristics. We have justified the design decisions behind SPAND and presented a detailed design of SPAND and mechanisms for incremental deployment.

Initial measurements of a SPAND prototype show that it can quickly provide performance estimates for approximately 95% of transfers from a site. These measurements also show that 69% of these estimates are within a factor of 2 of the actual performance and 90% are within a factor of 4.

We believe that a number of techniques will improve the accuracy of SPAND's performance estimates.

1. As clients are modified to transmit their own Performance Reports, the accuracy of the reports will improve. This will in turn improve the quality of the estimates that the Performance Server provides.
2. The Performance Server currently returns the median of all past measurements as an estimate of future performance. The measurements presented in this paper were made over a relatively short time scale. As shown in Section 3.1, the distribution of network performance changes as time passes. To provide better estimates, the Performance Server must give newer Performance Reports greater importance and discard information from older reports.
3. The performance of many transfers is limited by the round trip time to the server instead of the available bandwidth. We can improve the quality of SPAND's performance estimates by providing round trip estimates as part of the service and using both throughput and round trip times to predict the duration of a transfer.
4. The Performance Server currently combines the reports of all clients within its domain. It makes no attempt to eliminate poorly configured or misbehaving hosts. Preventing these hosts from impacting the estimates of network performance should reduce persistent sources of error.
5. The Performance Server estimates performance to all IP addresses outside its domain independently. However, network performance to many remote hosts is identical since communications to these hosts share the same bottlenecks. For example, the performance of most connections between the United States and Europe is probably limited by a shared transatlantic bottleneck link. We plan to add *Aggregation Experiments* to the Performance Server that allow it to analyze the distribution of reports to remote hosts over time and combine distant hosts into classes of equivalent connectivity. The server can then use reports from one of the hosts in a class to make or update estimates about the connectivity of other hosts in the same class.
6. Currently, the Packet Capture Host only generates Performance Reports for the bulk transfer Application Class. We plan to modify the Packet Capture Host to generate Performance Reports for other Application Classes such as telnet and server program execution (CGI programs).

7. Acknowledgments

Thanks to Vincent Cina and Nick Trio for providing the access necessary to install SPAND at IBM. Hari Balakrishnan, Steve Gribble, Todd Hodes, and Venkata Padmanabhan provided many useful comments on earlier versions of this paper that helped improve the presentation. This work is supported by DARPA contract DAAB07-95-C-D154 and grants from the California MICRO Program, Hughes Aircraft Corporation, Ericcson, and IBM. Mark is supported by an IBM fellowship.

8. References

[1] M Arlitt and C.L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proc. ACM SIGMETRICS '96*, May 1996.

[2] H. Balakrishnan, S. Seshan, M. Stemm, and R.H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Proc. ACM SIGMETRICS '97*, June 1997.

[3] R. Barrett, P. Maglio, and D. Kellem. How to Personalize the Web. In *Proc. CHI '97*, 1997.

[4] J.C Bolot. End-to-End Packet Delay and Loss Behavior in the Internet. In *Proc. ACM SIGCOMM '93*, San Francisco, CA, Sept 1993.

[5] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Computer Science Department, Boston University, March 1996.

[6] R. L. Carter and M. E. Crovella. Measuring bottleneck-link speed in packet switched networks. Technical Report BU-CS-96-006, Computer Science Department, Boston University, March 1996.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings 1996 USENIX Symposium*, San Diego, CA, Jan 1996.

[8] Cisco Distributed Director Web Page. http://www.cisco.com/warp/public/751/distdir/index.html, 1997.

[9] P. Francis. *http://www.ingrid.org/hops/wp.html*, 1997.

[10] J. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Proc. Fifth IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[11] R. Hinden and S. Deering. *IP Version 6 Addressing Architecture*. RFC, Dec 1995. RFC-1884.

[12] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.

[13] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Transactions on Networking*, February 1995.

[14] libwww-perl-5 home page. http://www.linpro.bo/lwp, 1997.

[15] M. Mathis and J. Mahdavi. Diagnosing Internet Congestion with a Transport Layer Performance Tool . In *Proc. INET '96*, Montreal, Canada, June 1996.

[16] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter '93 USENIX Conference*, San Diego, CA, January 1993.

[17] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical Report 95/5, Digital Western Research Lab, October 1995.

[18] C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*. RFC, Nov 1993. RFC-1546.

[19] pathchar – A Tool to Infer Characteristics of Internet Paths. ftp://ee.lbl.gov/pathchar, 1997.

[20] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U. C. Berkeley, May 1996.

[21] Socks Home Page. http://www.socks.nec.com, 1997.

[22] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, Nov 1994.

[23] UC Berkeley Annex WWW Traces. http://www.cs.berkeley.edu/ gribble/traces/index.html, 1997.

Rate of Change and other Metrics: a Live Study of the World Wide Web

Fred Douglis[*]
Anja Feldmann[†]
Balachander Krishnamurthy[‡]
AT&T Labs – Research

Jeffrey Mogul[§]
Digital Equipment Corporation – Western Research Laboratory

Abstract

Caching in the World Wide Web is based on two critical assumptions: that a significant fraction of requests reaccess resources that have already been retrieved; and that those resources do not change between accesses.

We tested the validity of these assumptions, and their dependence on characteristics of Web resources, including access rate, age at time of reference, content type, resource size, and Internet top-level domain. We also measured the rate at which resources change, and the prevalence of duplicate copies in the Web.

We quantified the potential benefit of a shared proxy-caching server in a large environment by using traces that were collected at the Internet connection points for two large corporations, representing significant numbers of references. Only 22% of the resources referenced in the traces we analyzed were accessed more than once, but about half of the references were to those multiply-referenced resources. Of this half, 13% were to a resource that had been modified since the previous traced reference to it.

We found that the content type and rate of access have a strong influence on these metrics, the domain has a moderate influence, and size has little effect. In addition, we studied other aspects of the rate of change, including semantic differences such as the insertion or deletion of anchors, phone numbers, and email addresses.

[*]Email: douglis@research.att.com.
[†]Email: anja@research.att.com.
[‡]Email: bala@research.att.com.
[§]Email: mogul@pa.dec.com.

1 Introduction

The design and evaluation of Web server caches, and especially of caching proxy servers, depends on the dynamics both of client reference patterns and of the rate of change of Web resources. Some resources are explicitly indicated as uncacheable, often because they are dynamically generated. Other resources, though apparently cacheable, may change frequently. When a resource does change, the extent of the change can affect the performance of systems that use *delta-encodings* to propagate only the changes, rather than full copies of the updated resources [2, 12, 16]. The nature of the change is also relevant to systems that notify users when changes to a page have been detected (e.g., AIDE [7] or URL-minder [17]): one would like to have a metric of how "interesting" a change is. One example of an interesting change is the insertion of a new anchor (hyperlink) to another page.

A number of recent studies have attempted to characterize the World Wide Web in terms of content (e.g., [4, 22]), performance (e.g., [20]), or caching behavior (e.g., [11]). These studies generally use one of two approaches to collect data, either "crawling" (traversing a static Web topology), or analyzing proxy or server logs. Data collected using a crawler does not reflect the dynamic rates of accesses to Web resources. Data collected by analyzing logs can provide dynamic access information, such as access times and modification dates (although most existing servers and proxies provide meager log information, at best, and dynamically generated data will not typically include modification information).

To quantify the rate, nature, and extent of changes to Web resources, we collected traces at the Internet connections for two large corporate networks, including the full contents of request and response messages. One of these traces, obtained over 17 days at the gateway between AT&T Labs–Research and the external Internet, consists of 19 Gbytes of data. The other trace, obtained over 2 days at the primary Internet proxy for Digital Equipment Corporation, was collected by modifying the proxy software to record HTTP messages for selected URLs; it amounts to 9 Gbytes of data. The traces used in our study have been described elsewhere [16] and are discussed in greater detail in Section 2.

Our trace collection and analysis were motivated by several questions. A primary issue was the potential benefit of delta-encoding and/or compression to reduce bandwidth requirements, a study of which was presented separately [16]. Here we address other aspects of the rate of change. When possible, we consider how the metric is affected by variables such as frequency of access, content type, resource size, site, or top-level domain (TLD)[1]. We answer questions such as:

- How frequently are resources reaccessed? The frequency of reaccess is essential to the utility of caching and delta-encoding.
- What fraction of requests access a resource that has changed since the previous request to the same resource? If the fraction is high, simple caching may prove much less useful than a scheme that can take advantage of delta-encodings.
- How "old" are resources when accessed, i.e., what is the difference between the reference time and the last-modified time? The age of resources can be an important consideration in determining when to expire a possibly stale copy [11].
- For those references yielding explicit modification timestamps, how much time elapses between modifications to the same resource, and how do the modification rate and access rate of a resource interact? If a cache can detect modifications at regular intervals, it can use that information to improve data consistency.
- How much duplication is there in the Web? When one requests resource *X*, how often does one get something identical to resource *Y*, either on the same host or another one? Examples of such duplication include explicit mirrors and cases where a particular resource, such as an image, has been copied and made available under many URLs. The rate of duplication may be important to the success of protocols such as the proposed "HTTP Distribution and Replication Protocol" (DRP) [19], which would use content signatures, such as an MD5 checksum, to determine whether the content of a resource instance is cached under a different URL.
- Can we detect and exploit changes in semantically distinguishable elements of HTML documents, including syntactically marked elements such as anchors and other interresource references (i.e., HREF tags), and untagged elements such as telephone numbers and email addresses?

[1]We use Bray's classification of TLDs [4], such as educational, commercial, government, regional, and so on.

Our analyses show that over a period of over two weeks, many resources were never modified, others were modified often, and a significant fraction were modified at least once between each traced access. The rate of change depends on many factors, particularly content type but also TLD. On the other hand, the size of a resource does not appear to affect modification rates. A significant fraction of resources overlap within or between sites (i.e., different URLs refer to identical bodies). Our analysis of semantically distinguishable elements showed that some elements, such as telephone numbers, are relatively stable across versions; others, such as image references, are more likely to change from one version to another and in some cases are replaced completely on a regular basis.

The rest of this paper is organized as follows. Section 2 describes our traces. Section 3 elaborates on the metrics we have considered and reports our results. Section 4 discusses related work, and Section 5 concludes.

2 Traces

Both the Digital and AT&T traces stored full-content responses for a collection of outgoing HTTP requests from a corporate community. They did not log non-HTTP responses, and the AT&T trace also omitted HTTP transfers from a port other than the default HTTP port 80 (these constituted less than 1% of the data).

The AT&T trace was collected by monitoring packets through the AT&T Labs–Research firewall. All resources were logged, enabling us to consider the effects of content-type on the AT&T reference stream. The trace consists of 950,000 records from 465 distinct clients accessing 20,400 distinct servers and referencing 474,000 distinct URLs.

The Digital trace was gathered by a proxy server that passed requests through a corporate firewall. The proxy did not cache resources, though clients could. Due to storage constraints, the trace software filtered out resources with a set of extensions that suggested binary

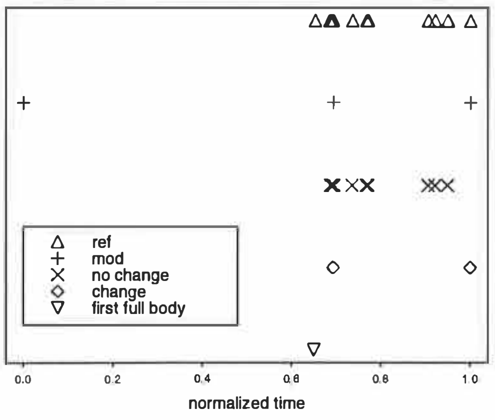


Figure 1: Visualization of the access stream for one resource. The x-axis represents a fraction of the period from the earliest last-modified timestamp for the resource until the latest reference to it. Each metric is spread across the y-axis (refer to the legend, and to the text for a detailed explanation.)

content, such as images and compressed data. Most, but not all, logged resources were textual data such as HTML.

Due to space constraints, we present results only for the AT&T trace. The restrictions on content-type in the Digital trace made it less useful for some of our analyses, but where we could obtain comparable results from both traces, we found them quite similar. The results from the Digital trace are available in an extended version of this paper [8].

3 Results

The following subsections correspond to the metrics discussed in the first section. Figure 1 provides a graphical representation of several of the attributes and their relationships to each other. Our metrics are derived from these attributes. For a particular resource, we consider a stream of accesses to it and the information available for each access. For status-200 responses (which return the body of the resource) and status-304 responses (which indicate that the resource has not changed since a previous time, provided in the request), we examine several attributes:

Request times The number of requests, and the time between each requests, is shown by the Δ marks in Figure 1.

Modification times The vast majority of status-200 responses (79%) contained last-modified timestamps (+ marks in Figure 1). When no last-modified information was available but the content changed, we assumed the resource was dynamically generated at the time of the request, and used the Date response header (or the timestamp of the local host if no Date header was provided).

Ages For those resources with a last-modified timestamp, the age[2] of a resource is the difference between the request time and the last-modified time. Otherwise, it is 0. In Figure 1, the age of each reference (Δ marks in Figure 1) is the difference between the timestamp of the reference and the modification timestamp immediately below or to the left of the reference.

Modification intervals To determine the interval between modifications, we must first detect modifications. The last-modified timestamp is not always present, and when it is present, it sometimes changes even when the response body does not. Therefore, we detect changes by comparing the bodies of two successive responses for a resource. The first time a full-body response is received, we cannot tell whether it has changed (∇ marks in Figure 1). Subsequent references are indicated as "no change" (**x**) or "change" ($\diamond$). For those modified responses with a last-modified timestamp, the time between two differing timestamps indicates a lower bound on the rate of change: the resource might have changed more than once between the observed accesses. If a modified response lacks a last-modified timestamp, then we assume that it changed at the time the response was generated. Again, the resource might have changed more than once between the observed accesses.

Statistics

In the following subsections, we present information about references and ages that span large time intervals—as much as 10^8 seconds (3.1 years) and higher. To focus on the trends across a wide time range, the graphs show the probability distributions with a logarithmic scale on the x-axis; the y-axis remains on a linear scale to emphasize the most common values. While a cumulative distribution function shows the probability that a value is less than x, it cannot clearly emphasize the most common values. Such values become more apparent when using a probability density of the *logarithm* of

[2]Note that this use of *age* differs from the HTTP/1.1 terminology, where a response Age header indicates how long a response has been cached [9].

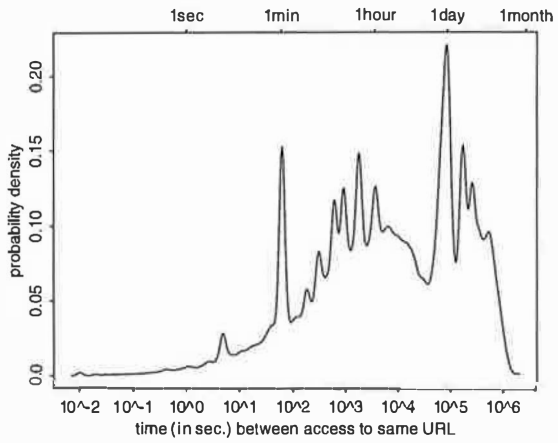


Figure 2: Density of time between accesses to the same resource, for all records in the AT&T trace. Time is shown on a log scale. Standard time units are shown across the top of the graph.

the data. Coupled with a logarithmic scale on the x-axis, plotting the density of the logarithm of the data facilitates direct comparisons between different parts of the graphs based on the area under the curve and is appropriate when using a log x-axis.

As we show later, content type bears on several of these statistics. Table 1 shows the distribution of the content types in the AT&T trace, as a fraction of unique resources and of all responses. In some cases a resource appeared with different content types over time, in which case the content type of the resource in our studies was determined by choosing the type that it appeared as most frequently. In terms of requests, images contributed to 69% of all accesses, and 64% of all resources. HTML accounted for just a fifth of accesses and about a quarter of resources. Application/octet-stream resources, which are arbitrary untyped data used by applications such as Pointcast, accounted for most of the rest of the accesses and resources. In terms of bytes transferred, GIFs contributed a relatively low amount of traffic for the number of accesses or resources, while all other content types contributed a greater share. (See [16] for additional statistics about content types.)

3.1 Access Rate

Figure 2 plots the density of the time between accesses to each resource for the AT&T trace. There are a number of peaks, with the most prominent ones corresponding to intervals of one minute and one day. The mean interarrival time was 25.4 hours with a median of 1.9 hours and a standard deviation of 49.6 hours. The huge difference between the median and the mean indicates that the mean is extremely sensitive to outliers. The mean of the data after applying a log-transform[3] gives a much better indication of where the weight of the probability distribution is. For this graph, the "transformed" mean is 1.6 hours.

Of the 474,000 distinct *resources* accessed in the AT&T trace, 105,000 (22%) were retrieved in a way that demonstrated repeated access: either multiple full-body status-200 responses, or at least one status-304 response that indicated that a client had cached the resource previously. A much higher portion of *references* (49%) demonstrated repeated access.

3.2 Change Ratio

We define the *change ratio* for a resource as the ratio of new instances to total references, as seen in the trace (i.e., the change ratio is the fraction of references to a resource that yield a changed instance). Overall we see that many resources are modified infrequently, but many more are modified often, and 16.5% of the resources that were accessed at least twice were modified every time they were accessed. Relative to all responses that were accesses more than once 13% had been changed since the previous traced reference to it. Yet considering all responses for which we could determine whether the resource was modified (either a status-304 response or a status-200 response that followed a previous status-200 response), 15.4% of responses reflected a modified resource.

Figure 3(a) graphs the cumulative fraction of resources that are at or below a given change ratio, organized by content type. Images almost never changed, while application/octet-stream resources almost always changed. For text/html, slightly over half the resources never changed, and most of the rest changed on each access after the first. However, this apparent high rate of change results largely from resources that were accessed just two or three times. Figure 3(b) shows just HTML resources, clustered by access count, and indicates that there is much more variation among the resources that were accessed six or more times, and that only about a fifth of those resources were modified on every access.

3.3 Age

Figure 4 presents density plots of the age of each resource when it is received, for those resources providing a last-modified timestamp. It omits resources for

[3]The log-transform of a set of data is exp(mean(log(data))).

Content type	Accesses		Resources	
	% by count	% by bytes	% by count	% by bytes
image/gif	57	36	48	18
text/html	20	21	24	33
image/jpeg	12	24	16	28
app'n/octet-stream	8	13	9	13
all others	2	6	3	8

Table 1: Content type distribution of the AT&T trace. Percentages do not sum to 100% due to rounding.

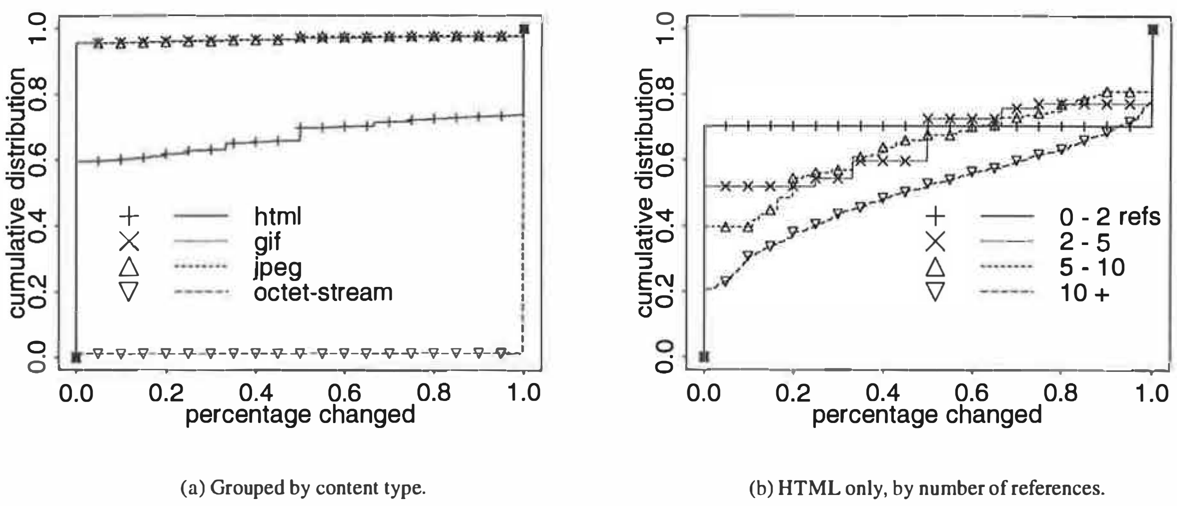


Figure 3: Cumulative distribution of change ratio for the AT&T trace.

which the modification date is the access time, such as dynamically-generated data and particularly a large number of application/octet-stream resources. The results are clustered in several ways:

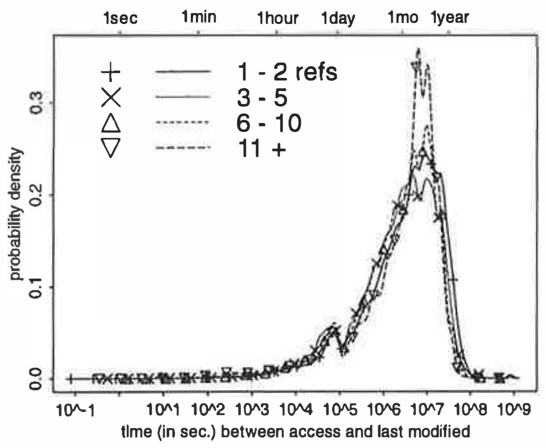
a. Resources are clustered by number of references. The most frequently referenced resources have the highest clustering of age, around the period of 10 days to 10 months. The curves are generally similar, indicating that the frequency of access does not have a large effect on age, although the most frequently accessed resources have a higher peak followed by a shorter tail (indicating somewhat younger responses overall).

b. Resources are clustered by content type, with application/octet-stream having a shape similar to the most frequent URLs in (a), since they contribute most of the references to the most frequently accessed resources. HTML responses are newer than the other types, with a mean of 1.8 months and a median of 12.8 days. This compares to a mean of 3.8 months and median of 63.9 days for gif resources, for example.

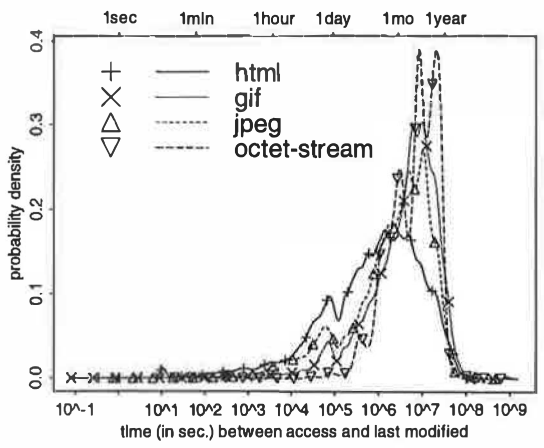
c. Resources are clustered by size. Here, there is not a large distinction between sizes, although the smallest resources tend to be somewhat older than others. This is unsurprising since there are many small images that are essentially static.

d. Resources are clustered by TLD, using Bray's categorization [4]. This clustering reduced the 20,400 host addresses to 13,300 distinct sites (such as a campus, or the high-order 16 bits of an IP address for numeric addresses). Educational sites serve resources that are noticeably older than other domains. Note that 17% of responses had no Host header; currently, these fall into the "other" category, and show some periodicity at an interval of 1 day.

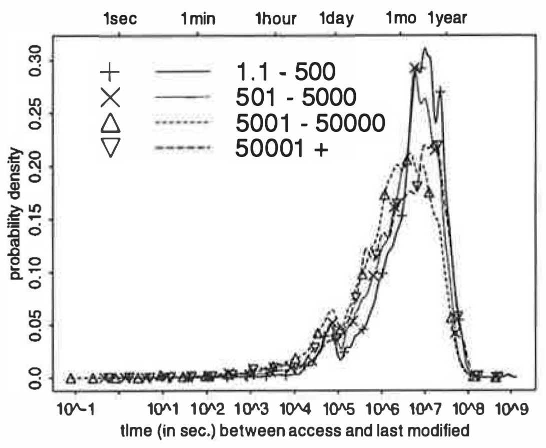
Previously, Bestavros [3] and Gwertzmann and Seltzer [11] found that more popular resources changed less often than others. As described next in Section 3.4, we found that when a resource changed, its frequency of change was greater the more often it was accessed. This result suggested that in our trace, more popular resources might change more frequently rather than



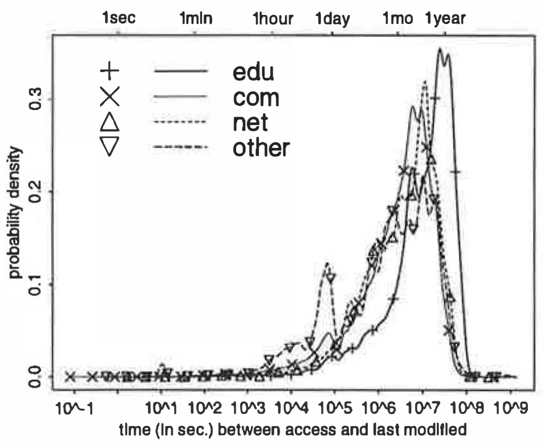
(a) Grouped by reference count.



(b) Grouped by content type.

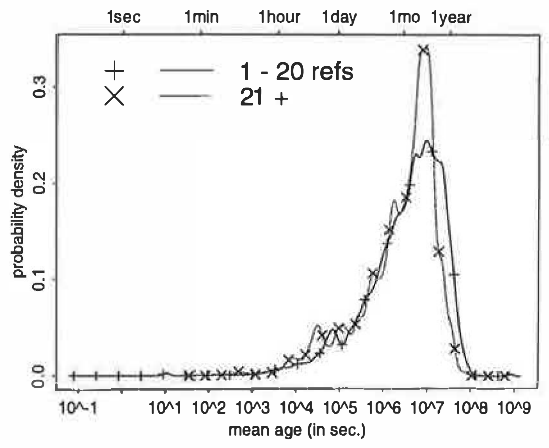


(c) Grouped by size (in bytes).

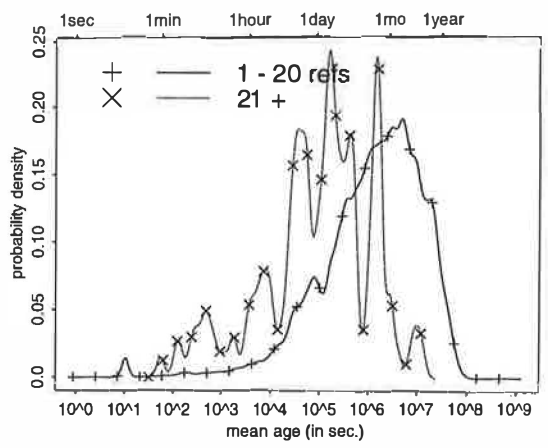


(d) Grouped by top-level domain (TLD).

Figure 4: Density plot of age of resources, clustered by various metrics, for the AT&T trace. Times are shown on a log scale.



(a) All content types.



(b) HTML only.

Figure 5: Density plot of age of resources, focussing on frequently accessed resources, for the AT&T trace. Times are shown on a log scale.

less. Figure 5 plots the mean age of resources, categorizing them into less frequently accessed resources (1–20 references) and more frequently accessed ones. Considering all content types (Figure 5(a)), more frequently accessed resources are clearly younger than less frequently accessed ones. Focussing on HTML (Figure 5(b)), the difference is even more pronounced.

The differences between our results and the earlier studies are striking, but they may be explained by considering the environments studied. The earlier studies reported on servers at Boston University and Harvard University (the educational TLD), while we looked at everything accessed by a community of users. A number of resources, such as "tickers" that update constantly changing information, were accessed frequently and changed on (nearly) each access.

3.4 Modification Rate

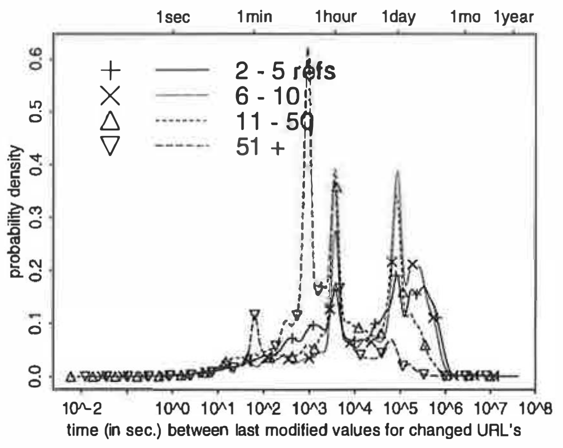
Figure 6 presents density plots of the time between last-modified timestamps for the AT&T trace, when a resource has changed. Figure 6(a) clusters the modification intervals by the number of accesses to each resource, demonstrating that, of the resources that change, the most frequently accessed resources have the shortest intervals between modification. (Naturally, since we are limited to observing modification times when resources are accessed, we cannot tell whether there are resources that are changing frequently but being accessed no more than once during the trace interval.)

The peaks in Figure 6(a) appear at somewhat intuitive intervals: 1 minute, 15 minutes, 1 hour, and 1 day. As the number of references increases, one is more likely to observe updates with fixed periodicity, such as the peak at 15 minutes for resources that are accessed 51 or more times. Also, the probability density falls off after 10 days, which is largely due to the 17-day duration of our trace and the need to observe two different last-modified dates.

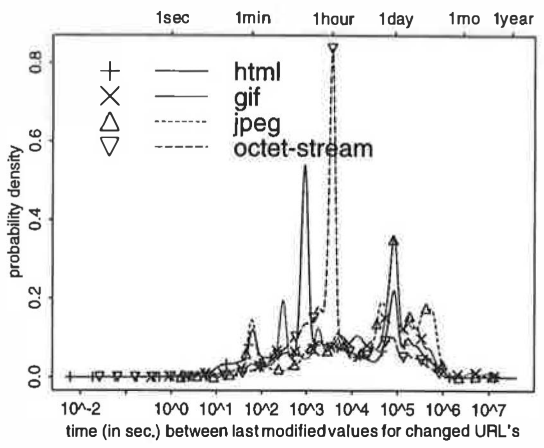
Figure 6(b) clusters the modification intervals by content type, and indicates that HTML resources that are modified at all will change more often than more static content types such as images. Some of these are the "tickers" mentioned above that are updated at 15-minute intervals. We also observe the effect of Pointcast (serving resources of type application/octet-stream), which has a one-hour update interval. Some images and HTML pages change daily, which may be partly due to the effect of advertisements and regular updates.

3.5 Duplication

Our content-based analysis required that we compute checksums of the content of each instance of each resource, in order to determine when changes occurred. In the process, we found some interesting phenomena: in particular, that 146,000 (18%) of the full-body responses in the AT&T trace that resulted in a new instance of a particular resource were identical to at least one other instance of a *different* resource. There are several common causes for this:



(a) Grouped by reference count.



(b) Grouped by content type.

Figure 6: Density plot of the time between last-modified timestamps, clustered by reference count and content type, for the AT&T trace. Times are shown on a log scale.

1. Multiple URLs may refer to a single server and return the same content. Most commonly this overlap is due to some form of unique session-identifier embedded within the URL. In one case alone, there were 443 distinct URLs that referred to the same content on the same host.

2. The same body may be replicated on multiple hosts, usually as an explicit "mirror," or an image that has been copied to reside with the HTML resources that embed it. The "Netscape Now" icon, the "blue ribbon" campaign, and various site-rating logos are examples of this.

3. Different resources may be used to convey information, for instance to inform a server of the origin of the link.

Figure 7(a) plots a cumulative distribution of the frequency of replication. Most bodies that are replicated appear just twice, but six appear over 400 times. Figure 7(b) plots the number of distinct hosts appearing in the set of resources for each replicated body, and shows that some appear just once (all replicas are served by the same host) while others follow the dashed line that indicates an equal number: every replica is served by a different host.

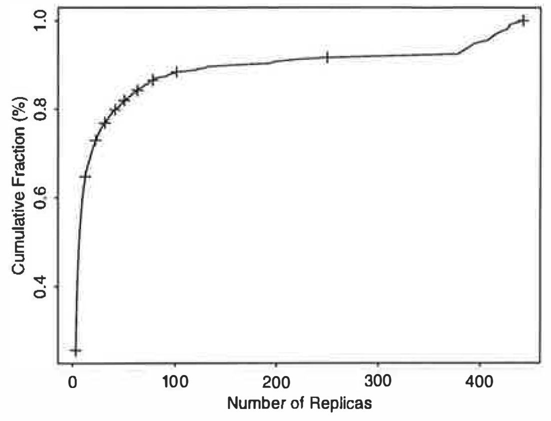
At first glance, the extent of replication suggests that a mechanism to identify replicas might serve to improve the effectiveness of caching. However, most of the resources are accessed multiple times and a traditional cache would eliminate many of the references to them. Of the rest, many are uncacheable and would need to be retrieved on each access regardless. Thus, the benefit of identifying duplicates would be to reduce the storage demands of the cache (not generally a large problem in today's environments) and to eliminate one access from each but the first host that serves the resource.

3.6 Semantic differences between instances

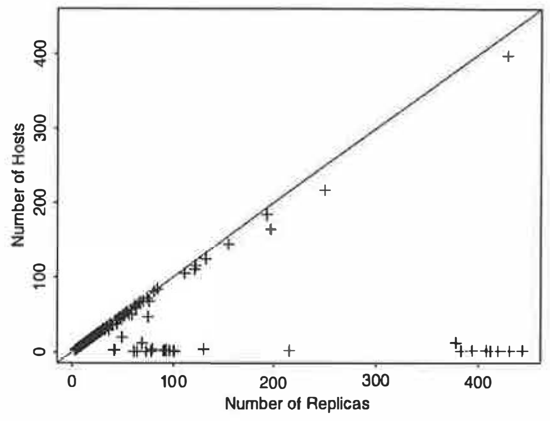
We define a semantically interesting item to be a recognizable pattern that occurs reasonably often in the context of Web pages. For example, telephone numbers (in various guises) are one class of pattern. Across instances of a Web page, changes in telephone numbers may be of interest. The manner in which we recognize semantically interesting patterns, referred to as *grinking* (for "grok and link"), is part of another project [14]; we concentrate here on the rate of change of semantically interesting items.

Using the AT&T trace, we looked for the following classes of patterns: HREFs (hyperlinks), IMGs (image references), email addresses, telephone numbers, and URL strings that occur in the body of a Web page. Because each of these forms can occur in many different ways, we probably did not recognize every occurrence. For example, a preliminary study [14] found over twenty different variations of North American telephone number syntax.

More importantly, we cannot always assert a string matching one of these patterns is indeed a telephone number. For example, it is possible that the string "202-



(a) Cumulative distribution of frequency of replication.



(b) Number of hosts by comparison to number of replicas.

Figure 7: Duplication of instances in the AT&T trace.

456-1111" is not actually a telephone number, although it is likely to be one, especially if the phrase "White House" appears in the same page. While we currently use a context-independent technique to recognize patterns, one could enhance the reliability of recognition by using the context surrounding the pattern. We would be more confident of a pattern suspected to be a telephone number if the string "phone", "telephone", or "tel" occurs in the surrounding context.

Grinking only makes sense for text files, which greatly reduced the number of responses we had to analyze. Also, we decided to look at only the first ten instances of a resource, since later changes are likely to follow the same behavior. We looked at 29846 instances of 8655 different resources. 55% of these resources were referenced twice; 71% were referenced three times or fewer; 90% were referenced 9 times or fewer. Table 2 presents the number of instances that had no recognizable forms of a particular type, such as HREFs.

For each instance of each resource we computed the semantic change by looking at the addition and deletions of forms. We define the *churn* for a given form as the fraction of occurrences of that form that change between successive instances of a resource. For example, if an instance of a resource has eight telephone numbers, and the next instance of that resource changes four of those telephone numbers, then the churn is 50%. We computed a churn value for each instance of a resource that contained the given form (except for the first one seen in the trace), then averaged the result over all instances, including the first. The results are in Table 3, which shows, for each class of form, the fraction of original occurrences of that form (as a percentage) that experienced a given amount of churn. For example, 1.5% of the recognized email addresses changed between instances in at least 75% of the cases.

As shown in Table 3, 5% of IMG references changed totally between instances, while fully qualified (10-digit) phone numbers changed the least. In 98% of the cases, when 10-digit telephone numbers were present, they did not change at all between instances.

These results are not too surprising. The stability of forms like telephone numbers may be useful in other contexts. In the future, we would like to compare the semantic difference between instances against a bitwise delta-encoding. Such a measure would tell us if the instances differ *only* in recognizably meaningful ways.

3.7 Additional Statistics

We analyzed the packet traces to compute statistics on a number of issues beyond the issue of the rate of change. In particular, we were interested in the presence of information in the HTTP request or response headers that would affect the cachability of resources: modification timestamps, authentication, cookies, pragmas, or expiration timestamps. Of the 820,000 status-200 responses, 650,000 (79.4%) contained last-modified times, without which browsers and proxy-caching servers will generally not cache a resource. Surprisingly, 136,000 responses (16.5%) involved cookies, while 48,500 (5.9%) had some form of explicit disabling of the cache, nearly all of which are from a Pragma: no-cache directive.

Form	Instances	Percent
HREF	7720	25.9
IMG	8331	27.9
Email	23795	79.7
10-digit phone	27531	92.2
7-digit phone	23788	79.7

Table 2: Number of instances which had no forms recognized

Churn	HREF	IMG	Email	10-digit Phone	7-digit Phone
100%	3.3	4.7	1.4	0.9	3.2
$\geq 75\%$	5.6	6.2	1.5	1.0	4.9
$\geq 50\%$	9.7	12.6	2.1	1.4	6.3
$\geq 25\%$	17.8	24.6	2.6	1.6	7.1
0%	41.2	48.6	96.5	98.0	90.2

Table 3: Percentage of instances having a given value of "churn" in semantically recognized forms.

4 Related work

One can gather data from a number of sources, both static (based on crawling) and dynamic (based on user accesses). Viles and French [20] studied the availability and latency of a set of servers that were found through web-crawling, primarily to ascertain when machines were accessible and how long it took to contact them. Woodruff, et al [22] used the Web crawler for the Inktomi [13] search engine to categorize resources based on such attributes as size, tags, and file extensions. For HTML documents, they found a mean document size of 4.4 Kbytes. Bray [4] similarly used the Open Text Index [18] to analyze a large set of resources. He found an mean resource size of 6.5 Kbytes and a median of 2 Kbytes. Bray's study primarily focussed on the relationships *between* resources, e.g. the number of inbound and outbound links.

Our traces represent dynamic accesses, so the sizes of resources that are actually retrieved by a set of hosts is expected to be different from the set of all resources found by a web-crawler. In our AT&T trace, the mean was nearly 8 Kbytes, with a median of 3.3 Kbytes. Our Digital proxy trace showed a mean of less than 7 Kbytes, and a median of 4.0 Kbytes. The response-body size differences between our two traces is due to the omission of certain content types from the Digital trace; these content-types show a larger mean, and a larger variance, than the included types [16].

Several studies have considered dynamic accesses, though they have not considered the frequency or extent of modifications. Cunha et al. [6] instrumented NCSA Mosaic to gather client-based traces. Those traces were then used to consider document type distributions, resource popularity, and caching policies. Williams et al. [21] studied logs from several environments to evaluate policies governing the removal of documents from a cache. Like us, they used logs from proxy-caching servers as well as *tcpdump*, but they examined headers only. They noted that dynamic documents that are presently uncacheable could be used to transmit the differences between versions. This idea was developed in more detail in WebExpress [12] and "optimistic deltas" [2]. A later study by Mogul et al. [16] quantified the potential benefits of delta-encoding and compression, using the same traces as we used for this paper. Arlitt and Williamson used server logs from several sites to analyze document types and sizes, frequency of reference, inter-reference times, aborted connections, and other metrics [1]. Here we considered many of the same issues, from the perspective of a collection of clients rather than a relatively small number of servers.

Kroeger et al. [15] recently studied the potential for caching to reduce latency, using simulations based on traces of request and response headers. They found that even an infinite-size proxy cache could eliminate at most 26% of the latency in their traces, largely because of the same factors we observed: many URLs are accessed only once, and many are modified too often for caching to be effective.

Gribble and Brewer [10] studied traces from a large collection of clients at U.C. Berkeley, gathered via a packet-sniffer like the one used for our AT&T trace. They examined a largely different set of metrics, such as access rates, locality of reference, and service response times.

Broder, et al. [5] analyze the *syntactic* similarity of files, using a web-crawler to create "sketches" of all accessible resources on the Web. These sketches can be used to find resources that are substantially similar. Such an approach might be an efficient way to find near-duplicates to which our work on semantic differences (and our previous work on delta-encoding [16]) is best applied.

5 Conclusions and Future Work

We have used live traces of two large corporate communities to evaluate the rate and nature of change of Web resources. We found that many resources change frequently, and that the frequency of access, age since last modified, and frequency of modification depend on several factors, especially content type and top-level domain, but not size.

Our observations suggest limits on the utility of simple Web caches. The assumptions upon which most current Web caching is based, locality of reference and stability of value, are only valid for a subset of the resources in the Web. Designers of advanced Web caches must confront the high rate-of-change in the Web, if they are going to provide significant latency or bandwidth improvements over existing caches.

In addition to the rate-based analysis, we performed semantic comparisons to multiple versions of textual documents and found that some entities such as telephone numbers are remarkably stable across versions. Semantic comparisons may prove useful in conjunction with notification tools [7, 17] as well as search engines, directories, and other Web services.

We are collecting a significantly larger trace dataset, to verify our conclusions here. We also intend to perform an extended semantic-difference study to locate minor changes in otherwise-identical web pages. We plan to investigate whether rate-of-change metrics can identify cases where it might be useful to pre-compute and cache delta-encodings at the server, and where prefetching of resources or delta-encodings might be beneficial.

Acknowledgments

Gideon Glass provided helpful comments on an earlier draft. We also thank the anonymous referees for their comments.

References

[1] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants (extended version). Technical Report DISCUS Working Paper 96-3, Dept. of Computer Science, University of Saskatchewan, March 1996. Available as ftp://ftp.cs.usask.ca/pub/discus/paper.96.3.ps.Z.

[2] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of 1997 USENIX Technical Conference*, pages 289–303, Anaheim, CA, January 1997. Also available as http://www.research.att.com/~douglis/papers/optdel.ps.gz.

[3] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In *Proceedings of the 12th International Conference on Data Engineering*, pages 180–189, New Orleans, February 1996.

[4] Tim Bray. Measuring the Web. In *Proceedings of the Fifth International World Wide Web Conference*, pages 993–1005, Paris, France, May 1996. Also available as http://www5conf.inria.fr/fich_html/papers/P9/Overview.html.

[5] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, April 1997. Available as http://www6.nttlabs.com/HyperNews/get/PAPER205.html.

[6] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 1996. Also available as http://www.cs.bu.edu/techreports/95-010-www-client-traces.ps.Z.

[7] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. The AT&T Internet Difference Engine: Tracking and viewing changes on the web. *World Wide Web*, January 1998. To appear. Also published as AT&T Labs–Research TR 97.23.1, April, 1997, available as http://www.research.att.com/~douglis/papers/aide.ps.

[8] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web. Technical Report #97.24.2, AT&T Labs–Research, Florham Park, NJ, December 1997. Available as http://www.research.att.com/library/trs-/TRs/97/97.24/97.24.2.body.ps.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, et al. RFC 2068: Hypertext transfer protocol — HTTP/1.1, January 1997.

[10] Steven D. Gribble and Eric A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the Symposium on Internetworking Systems and Technologies*. USENIX, December 1997. To appear.

[11] James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of 1996 USENIX Technical Conference*, pages 141–151, San Diego, CA, January 1996. Also available as http://www.eecs.harvard.edu/~vino/web/usenix.196/.

[12] Barron C. Housel and David B. Lindquist. WebExpress: A system for optimizing Web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Rye, New York, November 1996. ACM. Also available as http://www.networking.ibm.com/artour/artwewp.htm.

[13] Inktomi. http://inktomi.berkeley.edu, January 1997.

[14] Guy Jacobson, Balachander Krishnamurthy, and Divesh Srivastava. Grink: To grok and link. Technical Memorandum, AT&T Labs–Research, July 1996.

[15] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the Symposium on Internetworking Systems and Technologies.* USENIX, December 1997. To appear. Available as http://WWW.cse.ucsc.edu/~tmk/ideal.ps.

[16] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of SIGCOMM'97*, pages 181–194, Cannes, France, September 1997. ACM. An extended version appears as Digital Equipment Corporation Western Research Lab TR 97/4, July, 1997, available as http://www.research.digital.com/wrl/techreports/abstracts/97.4.html.

[17] Url-minder. http://www.netmind.com/URL-minder/URL-minder.html, December 1996.

[18] Opentext. http://www.opentext.com, 1997.

[19] Arthur van Hoff, John Giannandrea, Mark Hapner, Steve Carter, and Milo Medin. The http distribution and replication protocol. W3C Note, available as http://www.w3.org/TR/NOTE-drp-19970825.html, August 1997.

[20] Charles L. Viles and James C. French. Availability and latency of World Wide Web information servers. *Computing Systems*, 8(1):61–91, Winter 1995.

[21] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of SIGCOMM'96*, volume 26,4, pages 293–305, New York, August 1996. ACM. Also available as http://ei.cs.vt.edu/~succeed/96WAASF1/.

[22] Allison Woodruff, Paul M. Aoki, Eric Brewer, Paul Gauthier, and Lawrence A. Rowe. An investigation of documents from the WWW. In *Proceedings of the Fifth International WWW Conference*, pages 963–979, Paris, France, May 1996. Also available as http://www5conf.inria.fr/fich_html/papers/P7/Overview.html.

RainMan: A Workflow System for the Internet

Santanu Paul, Edwin Park, and Jarir Chaar

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

As individuals and enterprises get interconnected via global networks, workflows that scale beyond traditional organizational boundaries and execute seamlessly across these networks will become relevant. We address the problem of designing a scalable workflow infrastructure for the Internet that supports both flexibility in workflow participation and interoperability between heterogeneous workflow system components.

***RainMan** is a distributed workflow system developed in Java that lives naturally on the Internet. RainMan is a loosely-coupled collection of independent services that cooperate with each other rather than a monolithic system. Some of the useful features of RainMan are browser-based workflow specification, participation, and management, and dynamic workflow modification. The RainMan system is based on **RainMaker**, our generic workflow framework that defines a core set of well-defined interfaces for workflow components.*

1. Introduction

Workflow systems are essential to organizations that need to automate their business processes [Silver95]. The attraction of these systems is that they help organizations specify, execute, and monitor their business processes in an efficient manner over enterprise-level networks [FlMa, VisFlo, InConc, ActWork]. Workflow systems provide improved throughput and tracking of processes, and better utilization of organizational resources.

As individuals and organizations become interconnected via global networks such as the Internet, they will attempt to team up in various ways to share work and processes *across* traditional organizational boundaries. An Internet infrastructure that enables such interactions as workflows would be of great value; unfortunately, traditional workflow systems designed for centralized workflow execution do not lend themselves well to these new workflow applications. Internet-wide workflows require an infrastructure that supports decentralized workflow execution, workflow system interoperability, dynamic workflow modification, and low-cost workflow participation. The proprietary and monolithic design of current workflow systems makes it very difficult to address these requirements.

The *Workflow Management Coalition* (WfMC) has proposed a reference architecture and defined interfaces for vendors of traditional workflow systems to interoperate [WfMC]. The WfMC standard is a useful step in the direction of interoperability, however, its main shortcoming is that it promotes a monolithic workflow system architecture that is not flexible or scalable enough to address the needs of Internet-wide workflow applications which must necessarily be loosely-coupled.

We designed *RainMan* as a distributed workflow system for the Internet. The system comprises a collection of independent and lightweight services on the network. Our Java implementation uses open standards and Web browser-based user interfaces. We are using RainMan to experiment with a range of interesting features such as *dynamic workflow modification* which allows a workflow graph to be changed during execution, *disconnected participation* which allows participants, designers and administrators on the Internet to be infrequently connected to the network, and *downloadable workflow execution* which allows workflow subprocesses to be downloaded across the Internet on demand.

RainMan is based on *RainMaker*, our generic workflow framework that defines a core set of abstract interfaces for workflow components. The main purpose of RainMaker is to facilitate the design and implementation of flexible, interoperable, and scalable workflow system components.

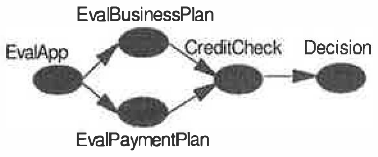


Figure 1: A Business Loan Approval Workflow

2. Workflow Systems Background

The traditional use of workflow systems has been in the automation of highly structured, process-based applications that are common to banks and insurance companies. Figure 1 shows a typical (simplified) *BusinessLoanApproval* workflow in a bank. Workflow systems provide extensive support for the design and automation of such applications in an efficient manner. Some of the important features they provide are:

1. Specification Tools: These are used to specify processes using high-level specification languages. There is no commonly-accepted basis for specification languages, each workflow system imposes its own language which can be based on directed acyclic graphs, rules, state machines, Petri-nets, or other any formalism. In general, it is common practice to program workflows visually as a directed graph where the nodes represent activities and the arcs represent control and data flows between activities.

2. Execution environments: A Workflow system provides an execution environment for workflow instances; this involves interpreting the process by stepping through the activities and assigning them to appropriate organizational resources such as humans, applications, and other workflow systems.

3. Audit Logs and Tools: These are necessary to monitor and track the progress of workflow instances through the workflow system.

4. Worklist Management: Management of *worklists*; worklists are persistent storage locations where human participants receive work assigned to them.

In effect, traditional workflow systems are sophisticated process specification and execution enviroments that allow organizations to run process-based applications efficiently, within the context of a closed environment over which the workflow system has significant authority.

In response to a growing number of proprietary workflow systems, the Workflow Management Coalition has defined a set of interfaces to enable workflow system interoperability. These interfaces are tied to a workflow system architecture known as the WfMC Reference Model (Figure 2) which defines interfaces between workflow servers and other components such as process definition or specification tools, workflow client applications, invoked applications, administrative tools, and other workflow servers. The specific details of these interfaces are described in [WfMC].

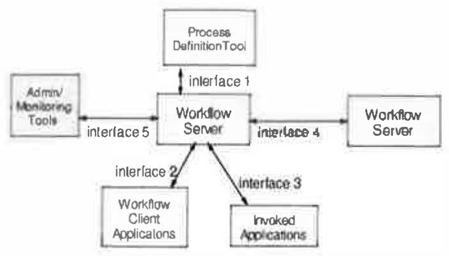


Figure 2: WfMC Reference Model

The architecture defined by the WfMC standard is not well suited for workflow execution on the Internet. The main problem is that the workflow server is monolithic. It is responsible for process execution, auditing, management of the organizational directory, and distribution of activities to appropriate participants (performing role resolution as necessary).

Most importantly, the server is responsible for hosting and managing worklists on behalf of all participants. This is problematic; since worklists are hidden within the workflow server (not externally addressable), activities can be sent only to worklists that reside in the same workflow server. Clients participating in multiple workflows on heterogeneous servers must have a worklist on each server and must manage each of the worklists (Figure 3).

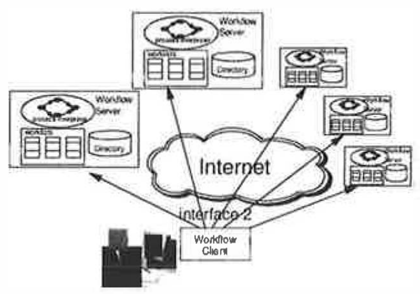


Figure 3: The Pull model

In addition, the workflow client application must manage multiple connections to each of these workflow servers since the architecture assumes a continuously connected workflow client. This is not a scalable or flexible solution, especially if one is to participate in a large number of workflows on the Internet, or use thin clients or PDAs, or operate in a disconnected mode. An analysis of the WfMC architecture reveals additional problems for decentralized workflow execution; interested readers may examine them elsewhere [Paula97, Schu96].

3. New Workflows on the Internet

In RainMan, we explore the potential of the Internet to enable decentralized workflow execution via interoperable workflow components that reside across this global infrastructure. We hope to enable new kinds of workflows involving dispersed individuals, multiple organizations, scattered network resources, and heterogeneous workflow systems. A few motivating examples of workflows that should be possible over the Internet are presented here.

Consider a virtual team of IT consultants from different organizations, scattered across the globe, working on a project that is coordinated via workflow. The consultants may be mobile and intermittently connected to the network. Irrespective of location, the project leader must monitor the workflow and work assigned to consultants must appear on their worklists. Each consultant may participate in many other workflows at the same time.

In such ***decentralized workflows***, participants must receive work from multiple workflows using heterogeneous clients ranging from desktops to laptops to PDAs (Figure 4). The work distribution model needs to be different from that proposed by WfMC. Instead of participants connecting to workflow servers explicitly to pull their work, the workflow infrastructure must automatically route work to them over a global network.

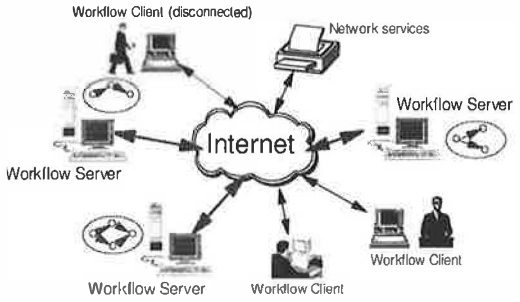


Figure 4: Decentralized Workflow Execution

Next, consider the *BusinessLoanApproval* workflow in Figure 1 running on a bank's workflow server. The *CreditCheck* step may itself be a nested subprocess that is delegated to a CreditEvaluation firm with a workflow server supplied by a different vendor. During execution, the bank's server would notify the firm's server to start an instance of its local *CheckCreditRatings* workflow. On completion of the subprocess, the bank's server would resume its suspended workflow (Figure 5). With global connectivity, these ***peer-to-peer workflows*** between heterogeneous servers are possible. In fact, the Interface 4 of the WfMC standard is designed to enable such workflow interactions. However, the drawback of the Interface 4 approach is the high cost of setting up such interactions, and significant pre-agreement required between participating servers.

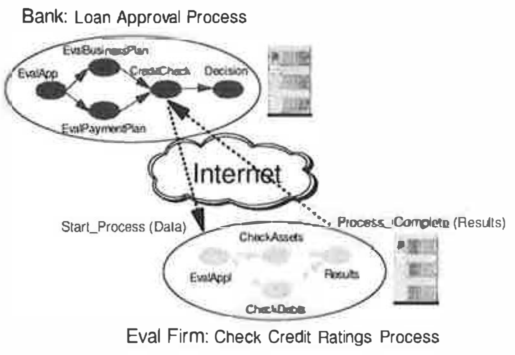


Figure 5: Peer-to-Peer Workflow Execution

Finally, as electronics commerce applications become available on the Internet, it is conceivable that workflows may be downloaded for just-in-time execution. ***Downloadable workflows*** make sense especially in cases where pre-installing and maintaining workflows is not cost effective. For example, consider an education brokerage service on the Internet [Hama96] that specializes in locating custom education services (Figure 6).

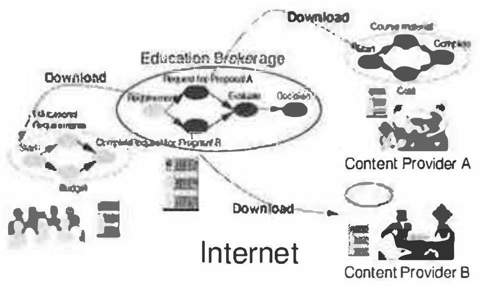


Figure 6: Downloadable Workflow Execution

In a plausible scenario, the *Brokerage workflow* downloads a *RequirementGathering* subworkflow to the client organization and requests its execution. Next, the *Brokerage workflow* downloads the requirements gathered to content providers along with a *RequestForProposal* subworkflow that the latter must execute to create the proposal. At the end, the brokerage compares all the proposals received and notifies the client.

To realistically enable workflows such as these on the Internet, significant issues related to security and organizational privacy must be addressed. Such services, based on state-of-art distributed systems security considerations, will have to coexist with the workflow infrastructure.

4. RainMaker Workflow Framework

RainMaker is a generic workflow framework we have developed to build inter operable workflow components. The details of the framework are available elsewhere [Paulb97]; only a brief outline is presented in this section.

RainMaker identifies four important abstractions within the workflow domain. Workflow instances are considered **Sources**, or service requesters. Sources generate **Activities**, or service requests that are delegated out. Instances of humans, applications, organizations, and other entities that handle these delegated requests are considered **Performers**, or service providers. Performers manage units of work called **Tasks,** which implement the delegated requests issued by Sources. It is an important characteristic of the workflow domain that Tasks can be long-running. A central feature of RainMaker is that a Task may be a workflow instance that recursively acts as a Source; this provides natural support for delegation of subprocesses.

The core RainMaker interfaces that help support this execution model are*:*

- *PerformerAgent:* An abstract interface that is implemented by Performers on the network. The interface provides mechanisms for delegating, controlling, and querying Tasks on the Performer.
- *SourceAgent:* An abstract interface implemented by Sources on the network. It provides a callback mechanism for Performers to return the results of Tasks to Sources.

In the rest of the paper, the italicized *SourceAgent* and *PerformerAgent* refer to the RainMaker abstract interfaces. The terms Source and Performer are used generically to refer to entities (or objects) that implement the RainMaker *SourceAgent* and *PerformerAgent* interfaces respectively.

The *PerformerAgent* and *SourceAgent* interfaces describe how proprietary Sources and proprietary Performers can interact with each other (Tables 1 and 2). In essence, the *PerformerAgent* interface conceals the internals of how a Performer or service provider actually performs Tasks in response to the Task requests. Symmetrically, the *SourceAgent* interface is a callback interface implemented to conceal the internals of how the Source actually generates Activities, issues Task requests, and handles responses from Performers. The interfaces also describe control mechanisms using which Tasks can be suspended, resumed, and aborted; and query mechanisms using which their status can be tracked. The interaction between Sources and Performers is shown in Figure 7.

PerformerAgent interface
`List(TaskDefinition)::listTaskDefinitions()`
`TaskID::createTask(SourceAgent source, ActivityID activityid, TaskRequest taskreq)`
`AbortID::abortTask(TaskID taskid)`
`SuspendID::suspendTask(TaskID taskid)`
`ResumeID::resumeTask(TaskID taskid)`
`TaskStatus::queryTask(TaskID taskid)`
`List(TaskID)::listTasks()`

Table 1: *PerformerAgent* interface

SourceAgent interface
`CompletedID::completedTask(PerformerAgent performer, ActivityID activityid, TaskResponse taskresp)`
`RefusedID::refusedTask(PerformerAgent performer, ActivityID activityid)`
`ForwardedID::forwardedTask(PerformerAgent performer, ActivityID activityid, PerformerAgent newperformer, TaskID taskid)`
`Boolean::seekPermissionToStartTask(PerformerAgent performer, ActivityID activityid)`

Table 2: *SourceAgent* interface

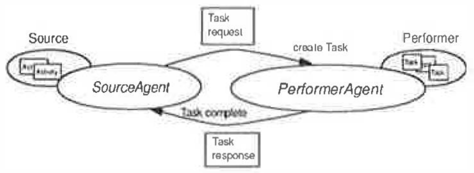


Figure 7: Source and Performer interaction

In addition to the core interfaces, RainMaker also defines the *Worklist* interface (Table 3). Worklists are an important workflow metaphor similar to electronic mail boxes; they provide a mechanism for human participants to access work assigned to them by workflow systems. In the case of RainMaker, Performer entities that represent human participants on the network can implement the *Worklist* interface and allow participants to view and access the Task Requests sent to them by various Sources.

Worklist interface
`List(WorkItemDescriptor)::getWorklistIndex()`
`WorkItemRequest::getWorkItem(WorkItemID wid)`
`void::completedWorkItem( WorkItemID wid, WorkItemResponse wresp)`
`void::refusedWorkItem(WorkItemID wid)`
`void::forwardWorkItem( PerformerAgent newperformer, TaskID taskid)`

Table 3: *Worklist* interface

The strength of the RainMaker framework lies in its generalized *push model* of Task distribution that allows Tasks to be delegated to Perfonners *independent* of their implementation. This aids the inter operability of heterogeneous workflow systems and components. Implementations of the *PerformerAgent* interface can represent arbitrary service providers: humans, applications, roles, organizational units, and workflow systems. Implementations of the *SourceAgent* interface can embody heterogeneous workflow applications described as rules, as control/data flow graphs, and even non-workflow applications such as collaborations or human Sources.

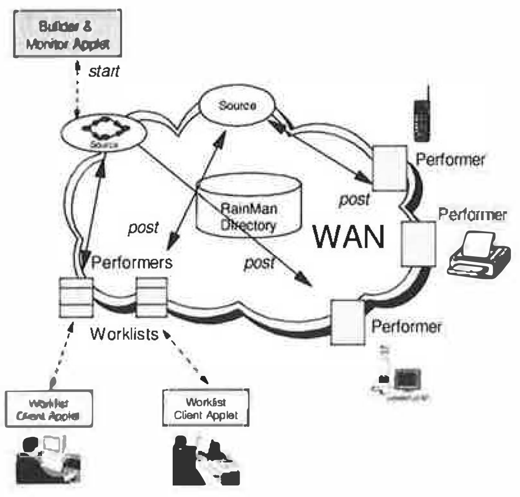


Figure 8: RainMan System Design

5. The RainMan System

The RainMan system (see Figure 8) is a distributed workflow system prototype written in Java using RainMaker interfaces. It consists of a loosely- coupled collection of distributed lightweight services required to deliver workflow functionality to Internet users. It currently consists of approximately 60 Java classes developed with Java JDK 1.1.4 and it uses Java's Remote Method Invocation (RMI) for transport between distributed components. The transport has been designed to be replaceable, it can be reimplemented on top of any messaging system. RainMan runs on a TCP/IP token ring network of Wintel machines.

5.1 RainMan User Interface Components

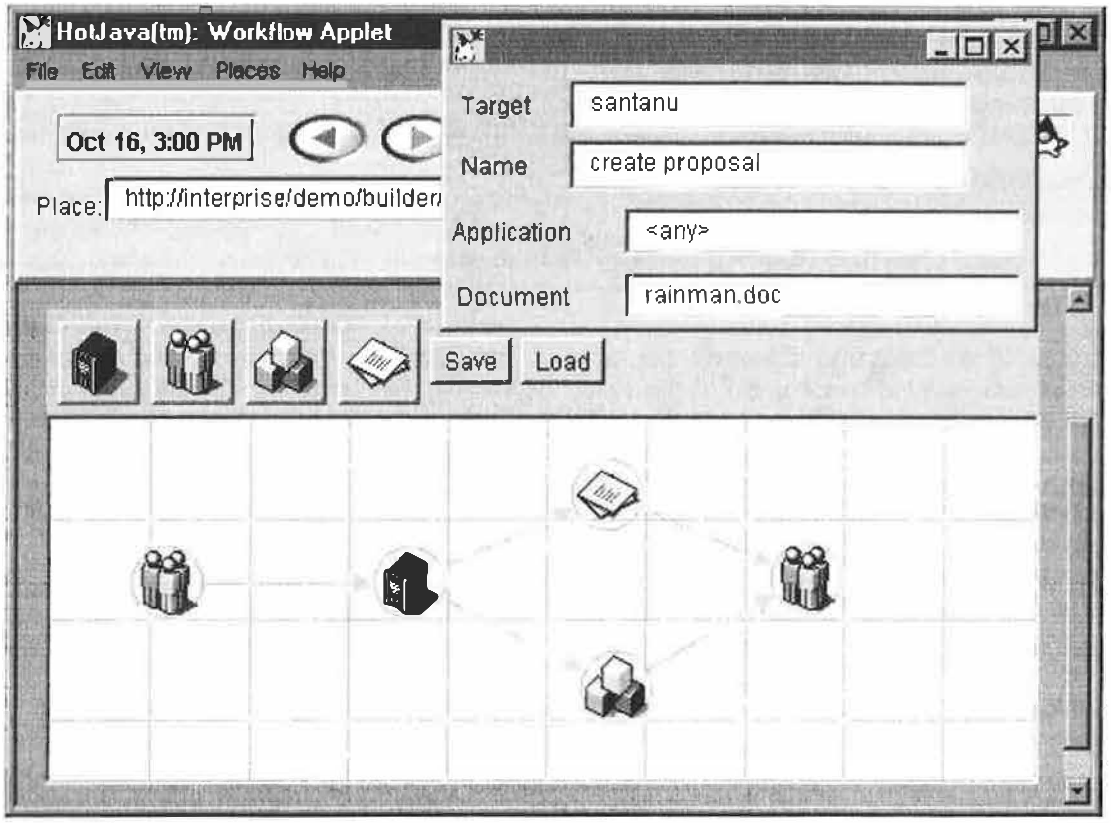
The user interface components of RainMan allow workflow users to interact with the runtime environment. The currently available ones are the Builder, the Worklist Client, and the Administrator, all implemented as Java applets. This makes any Java-compliant Web browser a usable client for all RainMan user interfaces, and has the additional benefit of making RainMan user interface components portable and hardware-independent. The Builder and Worklist Client are described in this section.

5.1.1 RainMan Builder Applet

The Rainman Builder (see Figure 9) in its current incarnation is a single Java applet that combines three functions. It is an interactive graphical environment for specifying workflows as directed, acyclic graphs. Perfonners are assigned from the RainMan directory service view available within the Builder. The service-specific aspects of an Activity are handled using the JavaBeans component object model. Workflow graphs can be saved to and loaded from a workflow specification repository.

The Builder also acts as a workflow graph interpreter (Source) and implements the *SourceAgent* interface. For an executing workflow, the Builder posts Task requests to specified Perfonners. On receiving notification of results, it inspects the workflow graph and posts the next set of Task requests. Since the Builder dynamically interprets the workflow graph, it is possible to change or refine the 'downstream' specification of the workflow graph even after the workflow has been started. This is a useful feature that enables the definition and execution of *ad hoc* processes, and distinguishes RainMan from traditional workflow systems. It allows for dynamic updates to the workflow graph to deal with emerging scenarios.

This is particularly relevant to decentralized workflow execution on the Internet, because we expect Perfonners to have significant autonomy over their internal domains and hence capable of raising exceptions in response to Task requests from Sources. In addition, Perfonners can change their capabilities as well as join or leave the network at will. Workflow systems that cannot be dynamically modified can be very limiting in such situations.



http://interprise/demo/builder/

Target: santanu
Name: create proposal
Application: <any>
Document: rainman.doc

Figure 9: Builder Applet

Support for groups and roles is an important part of workflow execution. Workflow systems are used to improve process throughput; one of the ways to achieve that objective is to assign an Activity to a role at build time. A role represents a group of 'fungible' Performers (i.e. 'designer' ' tester', 'manager', etc.); and the binding of an Activity to a specific Performer is postponed until execution. In RainMan, we are experimenting with the use of special-purpose Performers to manage roles; section 5.2.3 covers this topic in some more detail.

The choice of a good workflow specification language remains a matter of considerable debate within the workflow community. RainMan uses a vanilla workflow specification language based on directed, acyclic graphs. It is important to realize that RainMan offers an excellent infrastructure for deploying a wide range of specification tools based on heterogeneous languages, since the details (and potential idiosyncrasies) of the specification language are completely encapsulated within a Source implementation and not exposed either to the Performers or to the RainMan infrastructure. The clean separation of responsibilities of workflow routing and Task execution between Sources and Performers respectively makes RainMan a suitable infrastructure for plug-and-play operation with heterogeneous Sources and Performers.

Finally, the Builder also helps users monitor the state of a workflow execution. It provides visual cues to the owner or administrator about the global state of a workflow in execution at a coarse level of granularity.

Even though the three functions of Builder, Source, and Monitor are currently physically part of the same Java applet, they remain distinct logical entities in our workflow architecture. The next version of our prototype will separate some of these functions, allowing for features such as disconnected and remote building and monitoring.

5.1.2 RainMan Worklist Client Applet

The RainMan Worklist Client is a Java applet that allows users to access their Worklists from within a Web browser (see Figure 10). A Worklist is a remote Java object on the network that implements the *PerformerAgent* and *Worklist* interfaces and acts as the Performer for a human participant. The participant can use the Worklist Client to view the contents of the remote Worklist, and selectively download specific Task requests and perform them.

Appropriate Task Handlers are automatically launched to allow the human to interact with Task requests. Each participant or Performer has a set of capabilities in RainMan. For example, for humans capable of document processing, the Worklist Client currently handles incoming *Document_Processing* Task requests via a specialized Task Handler that can locate a referenced document from a network data store and launch the necessary local application (word processor) needed by the human to interact with the document. On completion, the Task Handler returns the response to its Worklist, which in turn returns it to the requesting Source.

Since each Task request received from the Worklist is handled locally on the client machine, disconnected operation is handled easily. The client applet needs to connect to its remote Worklist only for the purposes of receiving and returning Activities. However, a connection to the network may still be needed if the Task needs to reach certain data elements on the network in the course of its execution, or if the applications needed to perform it are not locally available.

The RainMan Worklist Client Applet is significant because it offers a valuable proof-of-concept that demonstrates how human performers can receive work from heterogeneous workflow sources using a single, unified user interface. Current workflow systems usually come with proprietary client applications that provide worklist access by pulling work from their proprietary servers. In contrast, RainMan requires that the Worklist Client pull work only from its designated Worklist on the network, to which Tasks requests are pushed from various workflow backends. The RainMan approach imposes a much lower burden on the Worklist Client since it no longer has to handle multiple connections, one with each workflow server it is receiving work from. In an inter operable workflow world, an equivalent of the RainMan Worklist Client Applet could replace proprietary worklist clients.

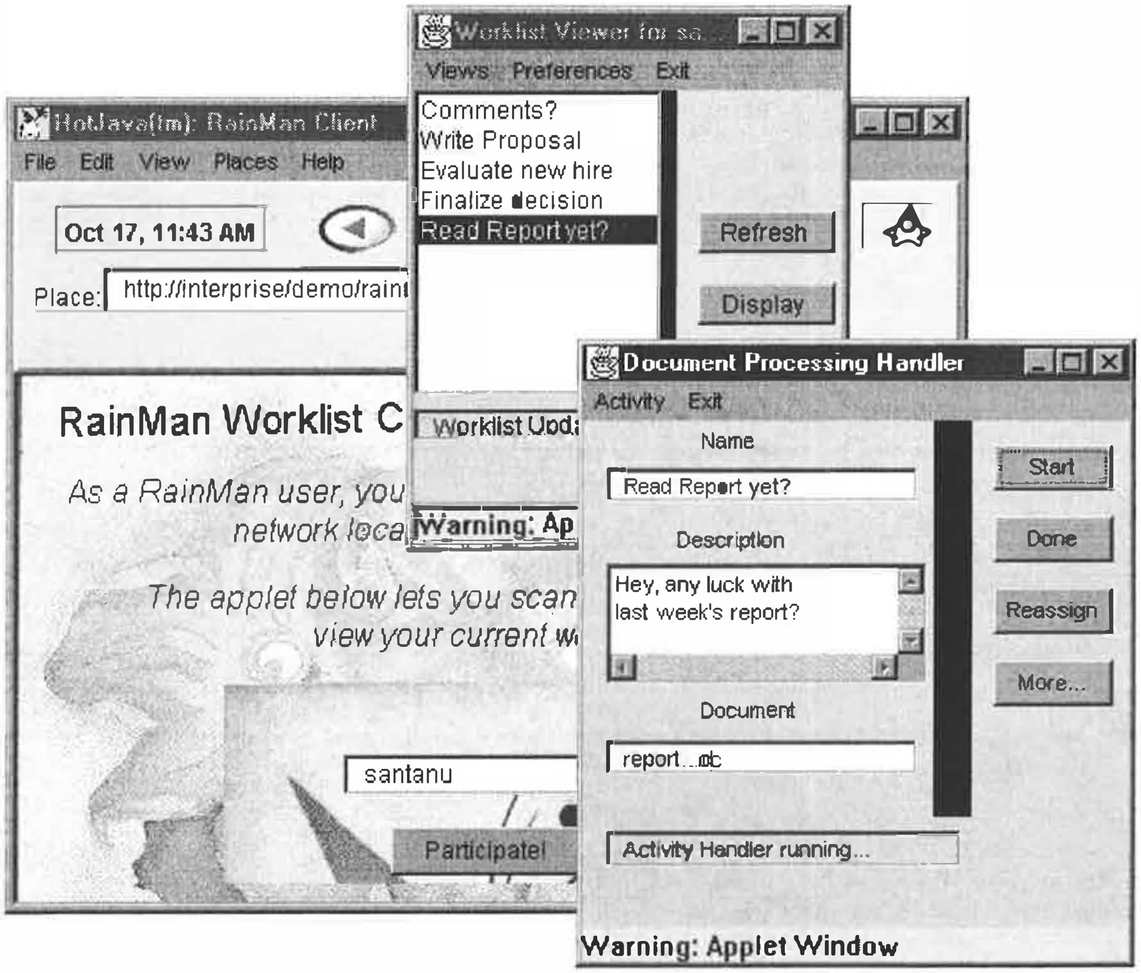


Figure 10: Worklist Client Applet

Task priorities and deadlines, while not yet implemented in RainMan, can be supported. Such constraints, *based on convention and agreements* between Sources and Performers, can be specified as part of the Task request message sent to a Performer. A Performer unable to meet these constraints may refuse to service the request. Similarly, a Source may use the deadline information associated with an Activity to time out on a Performer that does not respond within the deadline. However, attempts to generalize the behavior of arbitrary Sources and Performers with respect to priorities and deadlines leads to the broader question of negotiation protocols between Sources and Performers. Such protocols, while immensely useful, have not been studied in the context of the RainMaker framework.

5.2 RainMan Runtime Environment

The RainMan runtime environment provides a collection of distributed services that run on the Internet infrastructure. The user components described in the previous section allow workflow users - workflow designers, participants, and administrators - to interact with these distributed services on the network. The services currently available are the Directory Service, the Worklist Service, and a variety of Performers.

5.2.1 RainMan Directory Service

The RainMan directory service plays the key role of a trading service in the RainMan system. It contains information about Performers on the network and their capabilities. When a Source needs to issue a Task request to a Performer, it first performs a lookup operation on the directory service to locate the appropriate Performer on the network. The directory service interface also provides methods to register and unregister Performers. Using the Administrator applet, a RainMan administrator can manage the directory service and its contents. Mobile Performers use the directory service to locate their Worklists as well, thus enabling true location-independent workflow participation. For expediency, the RainMan directory service is currently prototyped as a custom Java application. We are planning an LDAP-based [Yeong95]directory service for the next version of the RainMan prototype.

For widespread workflow deployment on the Internet, it is necessary for the RainMan directory service to be distributed. The design of distributed directory services such as DNS [Mock87] and X.500 [CCITT]can be used as a guideline for RainMan directory design. RainMan directories in individual domains (a domain could be an organization, a geographical location, a collection of smaller domains, etc.) can be hierarchically arranged so that Sources can find Performers across domains. Performers within a domain would always be registered with their local RainMan directory. Sources would always lookup their local RainMan directories for Performers; the local directories could in turn access other RainMan directories by traversing the directory hierarchy to locate matching Performers. A related example is that of the Corba Trading Service [OMG97]; multiple Corba Traders can be explicitly linked into a network for transparent navigation.

5.2.2 RainMan Worklist Service

In workflow systems, worklists are inboxes associated with humans. In RainMan, a Worklist is a Java object that implements the RainMaker *Worklist* and *PerformerAgent* interfaces. Therefore, a RainMan Worklist is a Performer that is owned by and represents a human on the network (this is not a limiting assumption - Worklists can easily represent applications as well). Worklists offer persistent storage, via FIFO queues, of Task requests posted to humans from Sources.

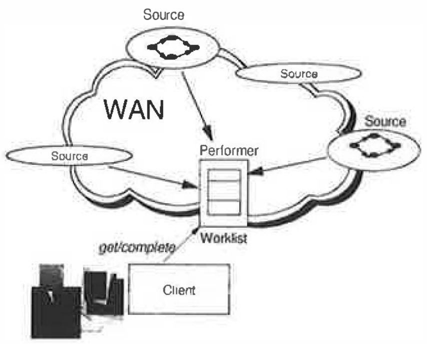


Figure 11: Reusable Worklists

In RainMan, Worklists are treated as addressable network objects, and provide a level of separation between Sources and actual human performers, which makes *asynchronous* exchange of requests and responses between them possible. In other words, requests can be posted to a Worklist on the network even when the human performer is not connected, and the human can access and perform them without connecting to the Source. Most importantly, in contrast to traditional workflow systems, a RainMan Worklist is a first class entity that can be reused by multiple Sources, thus eliminating the need for explicit, dedicated connections to workflow servers on the part of the performer (see Figure 11). This is in sharp contrast to the architecture shown in Figure 4.

Worklists on the network are managed by an independent RainMan Worklist Service. At the present time, the Worklist Service consists of a single Java application on the network, called a Worklist Server, that manages a large pool of Worklists. The Worklist Server is analogous to a POP [Myers96] or an IMAP [Cris93] server that stores e-mail boxes for multiple users. The clear separation of the Worklist Service from the Workflow Server is a novel design point in RainMan. This is useful because the Worklist Service is now streamlined to perform a dedicated function in an autonomous manner, and a Worklist Server can run on dedicated powerful computational resources that guarantees performance, availability, security, and reliability. Furthermore, since Worklists are practically independent of each other, we expect it to be relatively easy in this architecture to address scalability in terms of distributed workflow participants by implementing the Worklist Service as a distributed service; new Worklist Servers can be added to the network to host Worklists as the number of participants increases. We plan to experiment with these issues in the near future.

5.2.3 RainMan Performers

The Performer abstraction is useful in modeling humans, software applications, groups and roles, workflow servers, and entire organizations that perform Tasks on behalf of a workflow (see Figure 12). As we have seen, Worklists acts as Performers for humans. In addition, we are building a host of other Performers that can be used by RainMan workflows. An interesting Performer class is the `SMTPGatewayPerformer` that allows RainMan workflows to send out e-mail over the Internet. This Performer is an SMTP client written in Java that implements the *PerformerAgent* interface. It can receive *SendEmail* Task requests from Sources, connect to a SMTP server, and submit outgoing e-mail requests. Another implemented Performer class is the `DatabaseQueryPerformer` that can receive SQL queries from a Source and interface with a relational database at the back end via JDBC. A `PalmPilotPerformer` class has also been implemented that allows Worklist access from a US Robotics Palm Pilot.

With the growing use of cell phones, pagers, and PDAs, it is conceivable that a Performer and its human participant may communicate via other metaphors such as *publish/subscribe* (also known as *observable/ observer* or *model/view*), and its variations. The Performer and the human may be co-located on a single machine, or communicate over distances via a wide variety of networks (e.g., wireless, infrared, and so on).

In workflows, groups and roles are used to distribute Task requests to participants according to certain policies. The commonly supported scenario in current workflow systems is the case where an Activity is assigned to all members of a role, say insurance underwriters, and once a role member assumes responsibility for the Activity, it is retracted from the other role members. In RainMan, we take the view that a wide variety of distribution and retraction policies may be meaningful, depending on the application. For example, consider a company that wishes to post a *Request_for_Quotation* to each of its suppliers (these can be modeled as Performers). The company may wish to get results back from each of these suppliers before it can proceed with the next step in its workflow application. Alternately, it may just wish to receive responses from a 'majority' of its suppliers. We are designing concrete classes for Performers that implement such Task distribution and retraction policies based on roles; additional classes can be implemented based on the needs of specific applications.

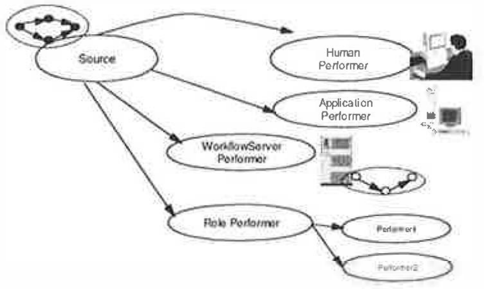


Figure 12: Heterogeneous Performers

6. Considerations in Workflow System Design

6.1 Performance

To the best of our knowledge, there are no published data on the performance and scalability of commercially available, proprietary workflow systems. However, with the increasing deployment of these workflow systems, concerns have been raised about the lack of adequate performance and scalability in even 'production' workflow systems (an industry label for workflow systems that specifically address the automation of high-volume, highly repetitive processes such as those in banks and insurance companies).

Our view is that the Internet will further aggravate the performance problems of centralized workflow architectures as they try to inter operate. WfMC-compliant, monolithic workflow servers are designed to handle process management, worklist management, auditing, and directory services simultaneously for hundreds of workflow instances at any given time. This architecture may perform acceptably as long as the number of participants is limited. However, the performance will degrade rapidly as the number of participants grows large.

The field of workflow urgently needs meaningful performance benchmarks that can be used to evaluate existing systems and their architectures. Until such benchmarks are established, it is not meaningful to quantitatively compare RainMan's performance with that of traditional architectures. However, we believe that a compelling qualitative argument in favor of RainMan can still be made. The RainMan design dismantles the traditional monolithic server into a collection of related components. This can help in eliminating the potential performance bottlenecks of workflow servers. For example, the clear separation between process management (Source-side) and worklist management (Performer-side) will allow each of these to be optimized independently. As more human Performers join the RainMan system, additional Worklist Servers can be pressed into service at different parts of the network. In a traditional system, additional worklists would all be added on the central workflow server, thus burdening the server itself. In contrast, the RainMan approach has no impact on the Sources and their performance.

6.2 Failure Handling and Compensations

Much of workflow research in recent years has focused on the transactional aspects of workflows [Rus94, Al96, Ley95]. The basic objective is to ensure the recoverability of workflows, since workflows are long-running applications that can execute over days or weeks or even months. It is fairly well-accepted in database transactions literature that classical flat ACID transactions are not viable in the context of long-running applications. Workflow researchers have thus borrowed concepts from nested transactions, sagas, and spheres of compensation to address the needs of workflows. While many of these ideas remain untested in commercial systems, they appear to be viable in the context of commercial workflow systems of today where the workflow server can exercise significant control over workflow participants.

The decentralization of the workflows, as proposed in RainMan, alters the picture significantly. In particular, if Source workflows are to execute on the Internet, the autonomy of Performers and their non-proximity to each other and to the Source itself must be taken as a given. A Source in this context has no global authority or jurisdiction over remote, heterogeneous, autonomous Performers, and has very limited visibility of their internal resources and mechanisms. In effect, all notions of compensation must be addressed in terms of commitments between peers; in effect, the interaction between a Source and a Performer must be sufficiently rich to handle failures and provide compensations for past services as necessary. To use a classical example, a Source may be a Travel Booking workflow that interacts with a Performer such as a Hotel Server. The Hotel Server may provide two basic operations - *reserve* and *cancel* - where *cancel* is a compensation for a *reserve.* Because of a subsequent change in travel plans, the Source may return to the Performer with a *cancel* request on its past *reserve* request. The Hotel Server, within the limits of its service contract, would have to fulfill this request. The idea of long-running *conversations* between autonomous network entities that engage in business transactions has been explored in the Coyote project [Dan97]. The Coyote view that meaningful business transactions can occur despite limited authority of each participant is a useful one for the Internet; we are exploring how RainMan Sources and Performers can benefit from the idea of conversations.

6.3 Decentralized Execution

Traditional workflow systems are based on a model of centralized workflow execution; the workflow system is responsible for managing workflow coordination as well as activity execution by invoking resources or participants entirely within its scope of authority - applications, other workflow servers, or human worklists.

A diametrically opposite model of workflow execution that can decentralize both workflow coordination and activity execution has been proposed in the context of the Arjuna project [Ran97]. This execution model decentralizes the coordination of a process by installing 'task controller' objects in different domains that coordinate with each other to deliver workflow routing functionality. Each task controller is a workflow 'router' that understands a piece of the overall workflow graph. This execution model eliminates a central point of failure in a workflow; moreover, workflows can proceed even in the face of partial network failures. The main consequence of the Arjuna approach is that decentralized workflow control requires participant domains (i.e. service provider domains) to participate in workflow routing on behalf of the workflow using a pre-agreed coordination language (a workflow routing protocol); this imposes computational burdens on participant domains that would be unacceptable if the domains are autonomous. Decentralized control can also be expensive to manage; it is harder to maintain global state and make dynamic changes to the workflow when the workflow script itself is decentralized.

The RainMan execution model strikes a middle ground. It separates the responsibility of workflow coordination from activity execution by creating two classes of entities, Sources and Performers. In effect, while the coordination of each process remains localized within a Source object, the actual execution of activities is decentralized across a network of Performers over which Sources have very limited control. The leverage in this model arises from the ability of heterogeneous Sources to share heterogeneous, autonomous Performers. This approach respects the autonomy of each Performer, assumes that the environments in which Sources and Performers execute will necessarily be heterogeneous, and makes it easier to keep track of global state as well as support dynamic workflow modifications.

6.4 Security Considerations

For workflows to run across wide area networks and especially across organizations, multiple security concerns must be addressed. First, an authentication mechanism must exist to validate the identity of both Source and Performer domains. This would allow basic functions such as Worklist access and Performer invocation to be done in a secure fashion only by authorized users or components. Second, access control rights need to be described and enforced in a scalable fashion to control access to methods on Performers and Sources. Third, the integrity and privacy of Task requests and responses exchanged between Sources and Performers should be maintained. Finally, support for nonrepudiability and enforcement of terms and conditions is needed. We are currently exploring these issues by drawing from the state-of-practice in distributed systems security. Many of these problems can be easily alleviated in the case of workflows between trusted parties by setting up private channels (Intranets or Extranets) between the participant individuals and organizations.

7. Conclusions

Our research is directed at designing an Internet workflow infrastructure that is scalable, flexible, and inter operable. This is a relevant and important problem since individuals and organizations are rapidly getting interconnected. This widespread inter connectivity can be exploited to enable new kinds of process-based applications.

The RainMaker framework defines the essential abstractions of a workflow system and facilitates interpretable workflow components. Using the RainMaker framework, we have implemented RainMan, a distributed object-oriented workflow system written in Java. Workflow management, activity distribution, directory services, and worklist management are all treated as independent services that work together to deliver workflow functionality to Internet users. This is a radical departure from traditional workflow systems based on monolithic, server-centric architectures. The RainMan system uses open standards and Web-browser based user interface components. The system is being used to experiment with a range of interesting features such as decentralized workflow execution, dynamic workflow modification, and disconnected participation.

While RainMan has been designed as an infrastructure for workflow execution, it offers insights into the broader problem of designing long-running applications on a network. In effect, RainMan highlights the importance of separating the responsibilities of service requesters (in this case, workflows) from service providers (in this case, humans, applications, organizations, etc.) via clean interfaces (i.e. *SourceAgent* and *PerformerAgent*), and assuming that entities that implement these interfaces are heterogeneous and autonomous. It offers a peer-to-peer Task delegation model with a nice recursive behavior; a Performer that receives Tasks requests from a Source can itself act as a Source of Tasks for other Performers on the network.

8. Acknowledgments

We thank David Hutches and Sastry Duri for various discussions on workflows and compensations. We also thank Carl Staelin and the reviewers of this paper for their valuable comments.

References

[ActWork] Action Technologies, Action Workflow, *http://www.actiontech.com*

[Al96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, and C. Mohan, Advanced Transactional Models in Workflow Contexts, In Proceedings of ICDE, 1996.

[CCITT] CCITT/ISO, X.500, The Directory - Overview of Concepts, Models and Services, CCITT/ISO IS 9594.

[Cris93] M. Crispin, IETF RFC 2060, Internet Message Access Protocol version 4 rev 1, December 1993

[Dan97] A. Dan, and F. Parr, The Coyote Approach to Network Centric Service Applications, 7th International Workshop on High Performance Transaction Systems, Asilomar, California, September 14-17, 1997.

[FlMa] IBM Corporation, FlowMark Workflow, *http://www.software.ibm.com/ad/flowmark/*

[Hama96] M. Hamalainen, A. B. Whinston, and S. Vishik, Electronic Markets for Learning: Education Brokerages on the Internet, CACM, Vol. 39, Number 6, June 1996.

[InConc] InConcert Inc., InConcert Workflow, *http://www.inconcert.com*

[Ley95] F. Leymann, Supporting Business Transactions via Partial Backward Recovery in Workflow Management, in Proceedings of BTW'95, Dresden, Germany, 1995, Springer Verlag.

[Mock87] P. Mockapetris, IETF RFC 1034/1035, Domain Names - Concepts and Facilities, Implementation and Specification, November 1987.

[Myers96] J. Myers, and M. Rose, IETF RFC 1939, Post Office Protocol - version 3, May 1996.

[OMG97] OMG Trading Object Service, CORBA Services: Common Object Services Specification, Chapter 16, July 1997.

[Paula97] S. Paul, E. Park, and J. Chaar, Essential Requirements for a Workflow Standard, OOPSLA Workshop on Business Objects Design & Implementation, October 6th,1997, *http://www.tiac.net/users/jsuth/oopsla97/*

[Paulb97] S. Paul, E. Park, D. Hutches, and J. Chaar, RainMaker: Workflow Execution Using Distributed, Interoperable Components, IBM Research Report nbr. 21008, October 1997.

[Ran97] F. Ranno, S.K. Shrivastava and S.M. Wheater, A System for Specifing and Coordinating the Execution of Reliable Distributed Applications, International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), September 1997.

[Rus94] M. Rusinkiewicz and A. Sheth, Specification and Execution of Transactional Workflows, In W. Kim, Editor, Modern Database Systems: The Object Model, Interopreability and Beyond, ACM Press, 1994.

[Schu96] W. Schulze, M. Bohm, and K. Meyer-Wegener, Services of Workflow Objects and Workflow Meta-objects in OMG compliant Environments, OOPSLA Workshop on Business Objects Design and Implementation, 1996.

[Silver95] B. Silver, The BIS Guide to Workflow Software, BIS Strategy Decisions, One Longwater Circle, Norwell, MA 02061, 1995.

[VisFlo] FileNet Corporation, FileNet Visual Workflo, *http://www.filenet.com/products/vwtext.html*

[WfMC] Workflow Management Coalition, http://www.aiai.ed.ac.uk/WfMC

[Yeong95] W. Yeong, T. Howes, and S. Kille, IETF RFC 1777, Light Weight Directory Access, March 1995.

Salamander: A Push-based Distribution Substrate for Internet Applications *

G. Robert Malan, Farnam Jahanian

Department of EECS
University of Michigan
1301 Beal Ave.
Ann Arbor, Michigan 48109-2122
{rmalan,farnam}@eecs.umich.edu

Sushila Subramanian

School of Information
University of Michigan
550 East University Ave
Ann Arbor, Michigan 48109-1092
sushila@umich.edu

Abstract

The Salamander distribution system is a wide-area network data dissemination substrate that has been used daily for over a year by several groupware and webcasting Internet applications. Specifically, Salamander is designed to support push-based applications and provides a variety of delivery semantics. These semantics range from basic data delivery, used by the Internet Performance Measurement and Analysis (IPMA) project, to collaborative group communication used by the Upper Atmospheric Research Collaboratory (UARC) project. The Salamander substrate is designed to accommodate the large variation in Internet connectivity and client resources through the use of application-specific plug-in modules. These modules provide a means for placing application code throughout the distribution network, thereby allowing the application to respond to network and processor resource constraints near their bottlenecks. The delivery substrate can be tailored by an application for use with a heterogeneous set of clients. For example the IPMA and UARC projects send and receive data from: Java applets and applications; Perl, C and C++ applications; and Unix and Windows 95/NT clients. This paper illustrates the architecture and design of the Salamander system driven by the needs of its set of current applications. The main architectural features described include: the data distribution mechanism, persistent data queries, negotiated push-technology, resource announcement and discovery, and support for Application-level Quality of Service policies.

*This work is supported by the National Science Foundation under the cooperative agreement IRI-92-16848 and a generous gift from the Intel Corporation.

1 Introduction

The availability of ubiquitous network connections in conjunction with significant advances in hardware and software technologies have led to the emergence of a new class of distributed applications. The Salamander data distribution substrate provides the systems support needed for two of these applications: groupware and webcasting. Participants in these wide-area distributed applications vary in their hardware resources, software support and quality of connectivity[11]. In an environment such as the Internet they are connected by network links with highly variable bandwidth, latency, and loss characteristics. In fact, the explosive growth of the Internet and the proliferation of intelligent devices is widening an already large gap between these members. These conditions make it difficult to provide a level of service that is appropriate for every member of a collaboratory or webcasting receiver.

Webcasting applications use push technology to send data from network servers to client machines. In group collaboratories, people from a global pool of participants come together to perform work or take part in meetings without regard to geographic location. A distributed collaboratory provides: (1) human-to-human communications and shared software tools and workspaces; (2) synchronized group access to a network of data and information sources; and (3) remote access and control of instruments for data acquisition. A collaboratory software environment includes tools such as whiteboards, electronic notebooks, chat boxes, multi-party data acquisition and visualization software, synchronized information browsers, and video conferencing to facilitate effective interaction between dispersed participants. A key challenge for the designers of wide-area collaboratories is the creation of scalable distribution and dissemination mechanisms for the shared data.

The Salamander substrate provides support for both webcasting and groupware applications by providing virtual distribution channels in an attribute-based data space. In a Salamander system, a tree of distribution nodes (servers) can be dynamically constructed to provide points of service into the data space. Clients can connect to this tree to both publish and subscribe to data channels. The data is *pushed* from suppliers to the clients through the distribution tree. Opaque data objects are constructed by clients that are described using text-based attribute lists. Clients provide persistent queries to the Salamander substrate using attribute expressions that represent the data flows they wish to receive, thereby subscribing to a virtual data channel. Salamander connections are first-class objects and are addressable if desired. This addressability allows for feedback from subscribers to data publishers. Additionally, Salamander allows for plug-in modules at any point in the distribution tree for application code to affect the data distribution. These plug-in modules provide the mechanism to support application-level Quality of Service policies.

Although the terminology is relatively new, Push technologies have been around for many years. The seminal push technology is electronic mail . Email has been pushed from publisher to subscribers for decades through mailing lists. Moreover, USENET news[9] has been used to push data objects (articles) based on text attributes (group names). Extending netnews, the SIFT tool[18] has been used to redistribute netnews articles based on text-based user profiles. At a lower level the combination of native IP multicast support and specific Mbone [5] routers provides a mechanism for the push of IP datagrams throughout portions of the Internet. Multicast datagrams are pushed based on a single attribute, namely the IP multicast group address.

Recently, many commercial push-based companies have started: BackWeb, IFusion, InCommon, Intermind, Marimba, NETdelivery, PointCast, and Wayfarer. These commercial ventures promise to manage the complexity of the web by providing data to users by means of subscription; similar to what current mailing lists provide, only more intrusively. In fact, many of these products are really *poll and pull* instead of push. The clients in these systems periodically poll the servers for new data, and then fetch it if it is available, reducing scalability.

The Salamander substrate differs from past technologies in several ways. First, it is not user-centric, but application-centric. Salamander is an underlying substrate that applications can plug into to transparently connect different portions of a wide-area application. This connectivity is achieved through the use of a channel subscription service. Second, Salamander allows for the addition of application plug-in modules along the distribution path to allow for a variety of data modifications and delivery decisions. The remainder of the paper will further describe the Salamander substrate. Section 2 provides background material on Salamander's current applications. Section 3 enumerates the main architectural features. Section 4 provides an overview of the Salamander application programming interface (API) as well as describes its administration and security features. Section 5 gives both a qualitative and quantitative performance evaluation. Finally, in Section 6 we conclude and describe our current and future research interests.

2 Application Domain

The Salamander substrate currently supports two applications with diverse needs: the UARC and IPMA projects. The Upper Atmospheric Research Collaboratory (UARC)[3, 17] is a distributed scientific collaboratory over the Internet. The UARC project is a multi-institution research effort, whose focus is the creation of an experimental testbed for wide-area scientific collaboratory work. The UARC system provides a collaboratory environment in which a geographically dispersed community of space scientists perform real-time experiments at a remote facilities, in locations such as Greenland, Puerto Rico, and Alaska. Essentially, the UARC project enables this group to conduct team science without ever leaving their home institutions. These scientists perform experiments on remote instruments, evaluate their work, and discuss the experimental results in real-time over the Internet. This community of space scientists has extensively used the UARC system for over three years; during the winter months, a UARC campaign – the scientists use the term *campaign* to denote one of their experiments – takes place almost every day. This community has grown to include regular users from such geographically diverse sites as: SRI International in Menlo Park, California; the Southwest Research Institute; the Danish Meteorological Institute; the Universities of Alaska, Maryland, and Michigan; and the Lockheed Palo Alto Research Laboratory.

The UARC system provides a variety of services to its users including shared synchronized displays for instrument data, multiparty chat boxes, a shared annotation database, and a distributed text editor.

However, the primary mechanism for collaboration is the real-time distribution of atmospheric data to the experiment's participants. This data is collected at remote sites such as Kangerlussuaq, Greenland, and is distributed over the Internet to the scientific collaboratory using the Salamander substrate described in this paper. Figure 1 shows several different data feeds displayed during a real-time campaign.

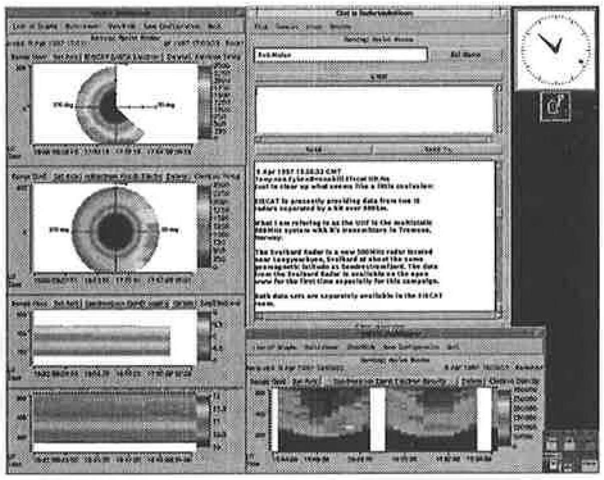


Figure 1: Example screen grab from April 1997 UARC campaign.

The second project that uses Salamander is the Internet Performance Measurement and Analysis (IPMA) project[8], a joint effort of the University of Michigan and Merit Network. The IPMA project collects a variety of network and interdomain performance and routing statistics at Internet Exchange Points (IXPs), internal ISP backbones, and campus LAN/WAN borders. A key objective of the IPMA project has been to develop and deploy tools for real-time measurement, analysis, dissemination and visualization of performance statistics. Two major tools from the IPMA projects are ASExplorer and NetNow. They both use Salamander to webcast the real-time data from the IPMA Web servers to their connected Java applets. ASExplorer is intended to explore real-time autonomous system (AS) routing topology and instability in the Internet. It supports measurement and analysis of interdomain routing statistics, including: route flap, growth of routing tables, network topology, invalid routing announcements, characterization of network growth and stability. An example of the ASExplorer client is shown in Figure 2. NetNow is tool for measuring network loss and latency. The NetNow daemon collects a variety of loss and latency statistics between network peers. The NetNow client is a Java applet which provides a graphical look at real-time conditions across an instrumented network.

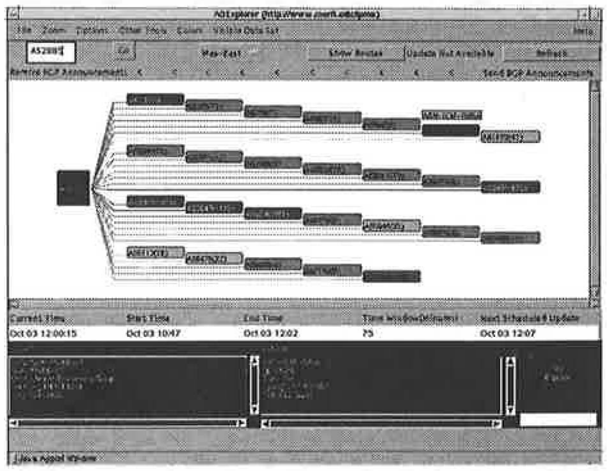


Figure 2: Example screen grab from ASExplorer session.

3 Architecture

The Salamander substrate's architecture can be described in terms of its coarse-grained processes, channel subscription interfaces, and distribution semantics. The key contributions of the architecture are its:

- **Channel Subscription Service**: The Salamander substrate uses a publish/subscribe service that combines the strength of database retrieval with a dynamic distribution mechanism. This service is used to provide a continuous flow of data from publishers to subscribers.
- **Application-level Quality of Service**: Application-level quality of service policies are supported that provide the ability to adapt the delivery channels in response to changes in client and network resources and subscriptions. These policies are supported by the use of application specific plug-in modules that can be used for data flow manipulation.
- **Lightweight Data Persistence**: Salamander employs the use of a caching and archival mechanism to provide the basis for high-level message orderings. A two-tiered cache keeps current data in memory while migrating older data to permanent storage. This storage takes the form of a lightweight temporal database tailored to Salamander's needs.

A Salamander-based system is composed from two basic units: *servers* that act as distribution points and are usually collocated with Web servers; and *clients* that act as both data publishers and subscribers. These units can be connected together in arbitrary topologies to best support a given application (see Figure 3 for an example). The Salamander server is designed from a utilitarian perspective, in that it can stand alone, or like a software backplane can be multiplied to increase scalability. The current version of the server is a POSIX thread implementation on Solaris. Salamander clients can both publish and subscribe to virtual data channels. In both the IPMA and UARC projects, the main data suppliers are written in either Perl or C; whereas the mainstay of the subscribers are Java applets. Applet development has progressed for over a year and a half on the UARC project, and Web browsers are the *de facto* subscriber platform. In the absence of multicast support in Java 1.0.2, and the lack of universal Mbone[5] connectivity, the decision was made to create our own distribution topology using Salamander servers in place of existing Mbone infrastructure. With the advent of Java 1.1 we are beginning to implement the Salamander interface using native multicast support.

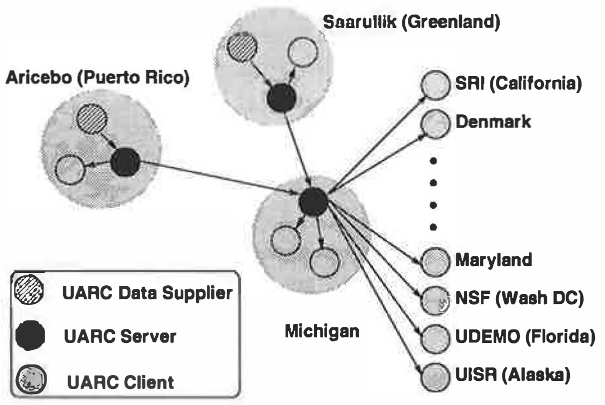


Figure 3: Example UARC campaign topology used during the April 1997 campaign

3.1 Channel Subscription Service

The Salamander substrate provides an abstraction for the distribution of data from publishers to subscribers through its channel subscription interface with both anonymous and negotiated push techniques. The basic idea in *anonymous push* is that publishers package opaque data objects, or Application Data Units as termed in [2], with text-based attribute lists. These attributes can then be used by Salamander to "lookup" destinations for the object. Subscribers place persistent *queries* to the Salamander substrate using lists of attribute expressions that can be used to match both current and future objects published to the Salamander space. Alternatively, this procedure can be thought of as accessing a distributed database where the queries are persistent. These persistent queries are matched by both the objects archived in Salamander's persistent repositories as well as future updates to the database space. These future updates and additions are dynamically matched with outstanding queries that are then pushed to the queries' corresponding clients. The query aspect of Salamander's attribute-based subscription service differs those in traditional nameservice and database systems, in that instead of the queries acting once on a static snapshot of the dataspace, they are dynamic entities that act on both the current state of the system and future updates. Publishers may come and go without affecting the connection between the Salamander database and the subscribers.

Salamander allows for feedback from subscribers to their publishers in the form of *negotiated push*. This is accomplished though the combination of unique endpoint addresses, endpoint namespace registration, and the ability to send unicast messages between Salamander clients. Each Salamander connection is given a unique address that it can insert into a global namespace. Connections manage their entries in this global namespace using the *supply* command. The `supply` command is given an attribute list, similar to the one used in the `query` command, that is paired with its identifier in the namespace. Other clients can then find entries in the namespace by matching the attributes with a *supply query*. Having obtained the identifier of an endpoint, the connection can then send it a *unicast* message. In practice, these sets of commands are used to denote the availability of data or membership in a group. In negotiated push, this mechanism can be used to allow subscribers to ask publishers to begin data distribution, or to modify a supplier's data flows at the source. The UARC application uses this mechanism to turn different data flows on and off. These data flows are computationally expensive, and should only be generated when there is a demand for the data.

A notification service is also provided within the Salamander namespace to allow for propagation of various system events to endpoints. For example, a connection can register a close event with their server that will cause the generation of a *close* notification message to a specified endpoint. In this way, clients can maintain membership information within their groups.

3.2 Application-Level Quality of Service

The Salamander architecture provides application-level Quality of Service policies to deliver data to clients as best fits their connectivity and processing resources[10]. These policies can rely on either best effort service or utilize network-level QoS guarantees[19] if available. Application specific policies are used to allocate the available bandwidth between a client's subscribed flows, providing a client with an effective throughput based on semantic thresholds that only the application and user can specify. These application-level QoS policies are achieved through the use of plug-in policy modules at points in the distribution tree. Figure 4 shows an example topology with several types of modules.

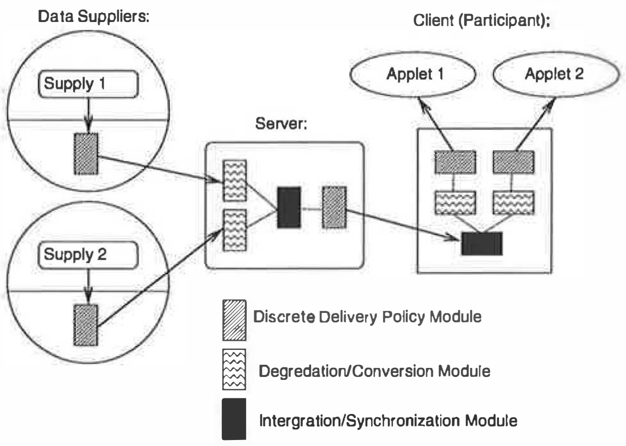

Data Suppliers:
Client (Participant):
Supply 1
Applet 1
Applet 2
Server:
Supply 2
Discrete Delivery Policy Module
Degredation/Conversion Module
Intergration/Synchronization Module

Figure 4: Example collaboratory topology with plug-in modules at suppliers, servers and clients. The architecture allows for the placement of data processing modules at any point in the distributed datapath. This specific example shows the path data takes from two suppliers to two client applets.

To illustrate the use of these modules, we use UARC as an example. Specifically, the UARC application uses discrete delivery, data degradation, and data conversion plug-in modules. By multiplexing the subscribed flows, discrete delivery modules can be used to prioritize, interleave, and discard discrete data objects. We have constructed a flexible interface that allows the client to both: determine its current performance level with respect to the supply, and to gracefully degrade its quality of service from an application-level standpoint so that it best matches the client connection's service level. These quality of service parameters are taken directly from the user in the UARC application. When users discover that they are oversubscribed, they can use the interface to specify priorities among the subscribed flows, and assign drop policies for the individual flows. A ***skipover*** policy can be used to specify fine-grained drop orders. Current skipover policies consist of both a FIFO ***threshold***, and a ***drop distance*** parameter. The threshold is used to allow for transient congestion; once this threshold of data has accumulated in the proxy, the drop distance parameter is used to determine which data are discarded, and which are delivered.

In addition to discrete delivery policies, the Salamander substrate provides for on-demand data degradation and conversion of data objects. In general, these mechanisms are used to convert one object into another. In order to support real-time collaboration between heterogeneous clients, some mechanism for graceful data degradation must be made available to provide useful data to the slower participants. At the application level, we understand something about the semantics of the shared data. We can exploit this knowledge, and provide a graceful degradation of the data based upon these semantics. The Salamander substrate uses on-demand (lossless and lossy) compression on semantically typed data, tailoring contents to the specific requirements of the clients. Lossless compression techniques are used for those data that cannot be degraded, such as raw text, binary executables, and interpreted program text. Lossy compression techniques are applied to data that can suffer some loss of fidelity without losing their semantic information, examples of which are: still and moving images; audio; and higher level text like postscript or hypertext markup languages. We give the application layer control over this quality by providing an interface that adjusts the fidelity of the real-time data flow on a per-client basis.

We were surprised by our experiences with the UARC project, in discovering that the system bottleneck was not network bandwidth, but was instead the processing power on the clients. The UARC system uses sophisticated Java applets to process raw scientific data as they arrive. During the April 1997 campaign, the Salamander substrate could deliver the UARC data to a large number of clients without difficulty; however, the clients' Java interpreters couldn't keep up with the incoming data rate. Our solution was to put plug-in modules in the distribution tree that could convert the data in-transit to a more manageable format.

Previous work has addressed the variability in client resources. Client resources are addressed in the Odyssey[13] system by sending different versions of the same object from the server depending on the client's resources. In [6], Fox et.al. provide a general proxy architecture for dynamic distillation of data at the server. The use of hierarchically encoded data distributed over several multicast groups is discussed in [12] for the delivery of different qualities of audio and video data. Balachandran et.al target mobile computing in [1] and argue for adding active filters at base stations. Active network proponents [15] argue that Internet routers should be enabled to run arbitrary application-level code to assist in protocol processing. The contribution of our work is the use of client feedback to allow for prioritization among flows; the construction of application and user interfaces for flow modification; and the ability to place modules at any point in the distribution tree.

3.3 Lightweight Temporal Database

Salamander provides data persistence by incorporating a custom lightweight temporal database. This database supports Salamander's needs by: storing a virtual channel's data as a sequence of write-once updates that are primarily based on time; and satisfying requests for data based on temporal ranges within the update stream. A temporal database [14] generally views a single data record as an ordered sequence of temporally bounded updates. In the Salamander database these records correspond to virtual channels. An administrator can determine which sets of virtual channels will be archived by the system. The Salamander system exports only a simple query interface to this database based on ranges of time and attribute lists. By foregoing the complexity of most commercial and research temporal databases, we could build a small and efficient custom database that met our simpler needs.

Salamander's synergy between real-time data dissemination and traditional temporal and relational databases is one of its significant contributions. It is taken for granted that queries on a relational or temporal database act as a static atomic action against a snapshot of a system's dynamic data elements. Our model alters this, by providing support for persistent queries that act over both a snapshot of the data elements present in the database and any modifications (real-time updates) to the database elements that may occur in the future. We plan to further address the impact of this model on database technologies in the future.

In addition to persistent state, Salamander maintains a memory cache used to buffer objects in-transit through the system. Together, the memory buffer and database act as a two-level cache that provides both high-level delivery semantics on the virtual flows and the ability to replay sections of flows from an archive. Salamander's virtual channels in the object space denote implicit groups. The delivery semantics within these groups varies depending on an application's needs. Groups can stay anonymous where the suppliers have no knowledge of the receivers, or they can become more explicit where the suppliers keep a tally of their receivers. A persistent disk-based cache of data objects, in conjunction with the use of application level framing[2], can be used to provide high-level delivery semantics. This includes: FIFO, causal, ordered atomic, etc[7].

4 Salamander Interfaces

There are two interfaces to the Salamander substrate: the application programmer interface (API) and the administration interface. The API has several layers and implementations, depending on the developer's needs. At the lowest level, the substrate provides a simple interface in both C and Java that gives four primitive operations: *send*, *receive*, *connect*, and *disconnect*. These operations are used in conjunction with a *property list* manipulation library to send and receive Salamander data objects. Much of this API can be illustrated by the example application code in Figure 5. This example is a small subroutine that subscribes to a simple virtual channel consisting of a single attribute.

The *connect* operation is demonstrated on line 8 of the listing, where a **`connectToSalamanderServer`** call is made. While **`hostname`** is straightforward, the **`sskey`** parameter requires some explanation. The **`sskey`** is a simple form of access control to the Salamander substrate. In the current implementation this key is very small, but one can imagine using a more robust key in conjunction with a secure socket implementation to achieve a greater level of confidence. Another security measure, implicit to the client, is an IP access list that restricts the access of data channels and administrative commands. This IP ACL is maintained on a per server basis.

After connecting to the substrate, the example then subscribes to the virtual channel by creating a query and submitted it to the server. This query is constructed in lines 14 through 19. The property list

```
void
subscribeToChannel(char * hostname, unsigned long sskey, char * queryName) {

    plist_t plist;
    void * dataptr;
    unsigned long data_length;

    if (connectToSalamanderServer(hostname, sskey) != SALAMANDER_OK) {
        fprintf(stderr, "Connection Error.");
        return;
    }

    /*  Create the query.  */
    plist = createPropertyList();
    updateProperty(plist, COMMAND_PROPERTY, QUERY_COMMAND);
    updateProperty(plist, NAME_PROPERTY, queryName);
    updateProperty(plist, COOKIE_PROPERTY, "queryCookie");
    sprintf(tmpbuf, "RANGE %d %d", begin, end);
    updateProperty(plist, TIMESTAMP_PROPERTY, tmpbuf);

    /*  Send it to the Server.  */
    if (salamanderSendServerData(plist, NULL, 0) != SALAMANDER_OK) {
        fprintf(stderr, "Error making query.");
        return;
    }
    destroyPropertyList(plist);

    /*  Read the responses as they come.  */
    for (;;) {
        if (salamanderReadServerData(&plist, &dataptr, &data_length) != SALAMANDER_OK)
            break;

        handleResponse(plist, dataptr, data_length);
    }

    disconnectFromSalamanderServer();
}
```

Figure 5: Simple C example that connects to a Salamander server and makes a single persistent query.

(`plist_t`) is the data structure that contains a data object's header and attribute information. Certain attributes are considered *well-known* by the system. An example, is the `COMMAND_PROPERTY` shown in line 15. The command property tells the substrate what to do with the data object upon receipt. In the example, the command is a query, which is intercepted by the substrate and is processed at the server. The `NAME` property is the only mandatory query attribute in the current implementation. While any number of attributes can be used to describe a virtual channel, one of them must be the `NAME` property. The `COOKIE` property on line 17 is used to match queries with a unique identifier that is returned by the Salamander substrate. This query identifier is added to any object that is returned by a subsequent *receive* operation, and is used to match responses with virtual channels. Finally, on lines 18 and 19, the *well-known* `TIMESTAMP` property is defined that designates the range of data for which the query corresponds. The *send* operation is carried out on line 22, followed by the destruction of the property list.

After the routine subscribes to the virtual channel, it goes into a loop that reads data objects from the substrate on lines 29 through 34. When the connection is severed, or another error occurs, the code *disconnects* and exits on line 36.

In addition to the base API, Salamander also exports an administrative interface. The substrate can be administered both remotely from a higher level API, and directly on the servers as configuration files. When used remotely, special commands are sent to the server using the base API. Example commands include: list current connections, list active virtual channels, prune a connection, resize a channel's memory buffer, establish a server peering connection, modify debugging level, etc. A server's configuration file is used primarily to establish static server peering relationships, to define default buffer sizes, and specify delivery semantics for specific virtual channels.

5 Performance Evaluation

The performance of the Salamander substrate can be characterized by both empirical and experimental results. Salamander is currently implemented as a multithreaded server on Solaris, and supports a set of complex Java applets that run on a variety of browsers[1,2]. Empirically, the Salamander substrate is used by both the UARC and IPMA projects as their base middleware for daily operation. Salamander has been used for over a year by the UARC scientists and has made a significant impact in the space science research community. The IPMA project has been using Salamander around the clock for over nine months for a variety of tasks. These tasks include: acting as the data collection mechanism for the NetNow probes at the Internet Exchange Points, and as the webcasting delivery mechanism from a central server tree to a collection of Java applets available from its website[8]. The UARC project has gone through several week-long campaigns using Salamander as the data distribution substrate that connects the remote instrument sites to the scientists' Java applets. During these campaigns over sixty scientists have had multiple connections receiving many different types of data. The empirical results from these two projects demonstrate that the Salamander system is both extremely robust and scalable.

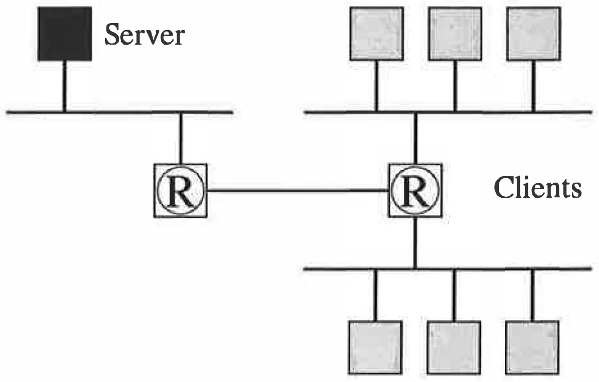


Figure 6: Experimental apparatus for Salamander server performance tests.

A series of experiments were performed on a single Salamander server to help quantify its performance. For all of the following experiments, the same setup was used. This corresponds to the network configuration shown in Figure 6. In this figure, the single Salamander server, shown in the upper left corner, is a SUN Ultrasparc-1 with a 140 MHz processor and 192 Mbytes of memory. A 10 Mbps Ethernet segment connects the server to a router

[1]Although counter-intuitive, cross browser compatibility is not a given. There are significant variations in Java VM implementations between browsers, and it was common during UARC applet development for code that would work under Netscape to break under Microsoft IE or HotJava (and vice versa).

[2]Coincidentally, HotJava continues to be the UARC developers' browser of choice for applet execution.

that is connected by a switching fabric to a second router. This second router has two interfaces that connect to university computing laboratories. Both of these laboratories consist of Ultrasparc-1 workstations connected by 10 Mbps Ethernet LANs.

The performance experiments highlight the scalability of the server in several dimensions: the number of simultaneous connections, and a throughput metric of objects per second. Both of these experiments characterize the substrate's performance in terms of a data object's end-to-end latency from a data supplier to a receiving client. To measure this latency, a timestamp is written into the object's attribute list as it flows down the distribution tree. These timestamps are written to a log file upon receipt at the subscribers. By analyzing the log files offline, the experiment's latency statistics can be extracted. Since the analysis of data in these experiments relies on this distributed timestamp information, a method for synchronizing the clocks on an experiment's hosts was applied. To compensate for the difference in the clocks, a probabilistic clock synchronization technique, similar to the protocols developed by Cristian [4] was used.

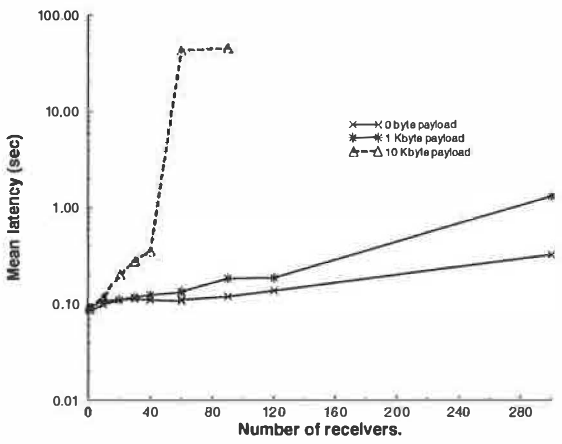


Figure 7: Results from a single supplier Salamander scalability experiment.

Figure 7 shows the results of experiments that evaluate the scalability of a single Salamander server in terms of both number of concurrent receivers and size of the data payload. Three payload sizes were used: 0, 1 Kbyte, and 10 Kbytes. However while the payload varied, each of the object's headers were filled with approximately 100 bytes of testing and channel information. For all of these results, a single data supplier was used that sent an object once per second for a period of five minutes. The horizontal axis represents the number of concurrent receivers during the experiment; whereas the vertical axis shows the mean delivery latency for the objects on a logarithmic scale. These results show that for a small data payload a significant number of clients can be supported. During the execution of these tests the available bandwidth between the server and the laboratories was approximately 400 Kbytes per second. This bandwidth was measured both informally using FTP latency and rigorously with the treno tool[16]. This explains the steep rise in the 10 Kbyte payloads between 40 and 50 receivers, that corresponds to 400 Kbytes and 500 Kbytes per second respectively. At levels of throughput above a link's capacity, Salamander's memory buffer fills and all latencies reach a steady state due to the finite buffer space and drop tail delivery semantics. A virtual channel's buffer space can be specifically tailored to bound this maximal latency of received objects.

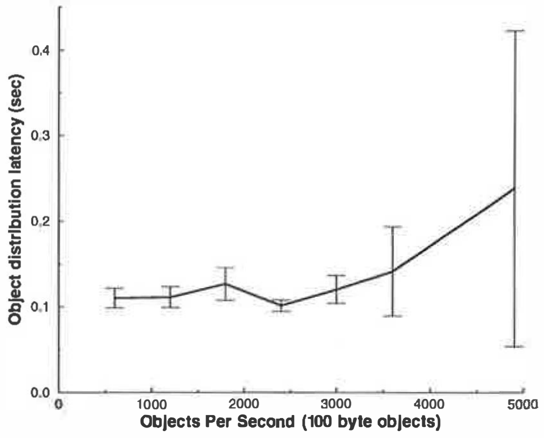


Figure 8: Measured latency of objects through the system when objects per second is varied.

The second set of experiments shows the scalability of a single Salamander server in terms of maximum objects per second (ops) that can be processed. This is similar to a datagram router's packets per second (pps) metric. The results of this set of experiments is shown is Figure 8. The data points in this graph represent the mean and standard deviation of the latency of objects from supplier to client under increasing server load. These points were generated by running experiments that sent an object with a header of 100 bytes and a payload of length zero from a set of senders to a set of receivers. Each sender sent a single object once per second. For example, in order to generate the 2400 ops data point a series of experiements were run using sets of 40 suppliers and 60 receivers. Unfortunately, the bandwidth between the server and the laboratories never exceeded 500 Kbytes per second for sustained periods while these experiments were undertaken. However, during the 4900 ops experiments the relatively slow 140 MHz processor was approximately 25% idle, leaving room for further scaling. The variation between the data points results from using an active testbed.

Together, these two sets of experiments show that a single server substrate scales with the available network bandwidth. Further improvements in scalability can be made by utilizing a heirarchy of servers in the substrate to offload bandwidth and processing overhead. The empirical results concur, showing Salamander to be both scalable and robust.

6 Conclusion

This paper presented both a functional description of the Salamander distribution substrate's architecture and interfaces; and a quantitative analysis of its performance characteristics. The contributions of the architecture are its combination of channel subscription service with a lightweight temporal database, and the definition and use of an application-level QoS framework.

The Salamander substrate's channel subscription service provides for both an anonymous and a negotiated push of data from a set of suppliers to a set of receivers. The support for anonymous push is straightforward, the suppliers know nothing about its set of receivers. In contrast, negotiated push support enables subscribers and publishers to negotiate the content of their data channels. To provide this functionality, Salamander includes: a registration and matching service, similar to a name service; and a notification service that propagates various system events, including client termination, to Salamander endpoints.

Application-level Quality of Service is defined and supported in the Salamander substrate as a way of tailoring the available resources to best fit the user and application. This is done by utilizing semantic knowledge that only the application has about its data and providing mechanisms for the graceful degradation of its virtual data channels. The specific contribution of our work is the use of client feedback to allow for prioritization among virtual channels; the construction of application and user interfaces for channel modification; and the ability to place channel conversion modules at any point in the distribution tree.

Salamander's incorporation of a lightweight temporal database provides the basis for a powerful synergy between real-time data dissemination and traditional temporal and relational databases. Our model provides support for persistent queries that act over both a snapshot of the data elements present in the database and any modifications (real-time updates) to the database elements that may occur in the future.

Salamander's research contribution is complemented by the utility and robustness of the current implementation. Several Internet applications were described that motivated Salamander's push-based approach to data distribution. These applications, namely the UARC and IPMA project applets, use Salamander around the clock to provide application connectivity throughout the Internet. Through day-to-day use, these applications have shown Salamander's empirical performance to be good. Moreover, the quantitative performance experiments show that a single server scales well for a significant number of connections; the server's bandwidth was the first-order bottleneck in these experiments.

Our current work is focused on enhancing the scalability of the system. Specifically, we are investigating the applicability of scalable routing technologies to Salamander's attribute-based data dissemination. Additionally, we are addressing several system administration aspects of the multiserver substrate, including pairing it with a key distribution mechanism. In concert with these activities, we plan to further address the impact of Salamander's persistent query model on database technologies. Finally, we are in the process of porting the server to both Java and Win32 based platforms, in order to objectively compare the base system in a variety of settings.

7 Availability

Additional information on the Salamander system can be found at the following URL:

`www.eecs.umich.edu/~rmalan/salamander/`

Information that can be found there includes:

- Binaries and source code for the Solaris version of the Salamander server,

- Generic C and Java interface libraries for both data suppliers and clients,
- Status of both a Java and Win32 port of the Salamander server.
- Information about ongoing research efforts based on Salamander.

Acknowledgments

Special thanks go to: Craig Labovitz the indefatigable Salamander evangelist; Jim Wan for his development and debugging aid; Mike Bailey for his constructive criticism; and Michael Willis for his initial efforts. Thanks also go to the entire UARC development team: Peter Knoop, Terry Weymouth, and Mike Burek. The performance experiments would have been unbearable without Don Libes's `expect`. Again we would like to acknowledge the National Science Foundation and Intel for their generous funding that made this work possible.

References

[1] Anand Balachandran, Andrew T. Campbell, and Michael E. Kounavis. Active filters: Delivering scaled media to mobile devices. In *Proceedings of the Seventh International Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.

[2] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM '90*, September 1990.

[3] C. R. Clauer et al. A new project to support scientific collaboration electronically. *EOS Transactions on American Geophysical Union*, 75, June 1994.

[4] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[5] H. Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8), 1994.

[6] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languates and Operating Systems*, October 1996.

[7] Vassos Hadzilacos and Sam Toueg. *Distributed Systems*, chapter 5, pages 293–318. Addison-Wesley, second edition, 1993.

[8] Internet Performance Measurement and Analysis (IPMA) project homepage. http://nic.merit.edu/ipma/.

[9] B. Kantor and P. Lapsley. Network news transfer protocol (nntp). RFC 977, February 1986.

[10] G. Robert Malan and Farnam Jahanian. An application-level quality of service architecture for internet collaboratories. In *Proceedings of the IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems*, December 1996.

[11] G. Robert Malan, Farnam Jahanian, and Peter Knoop. Comparison of two middleware data dissemination services in a wide-area distributed system. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[12] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM '96*, August 1996.

[13] Brian D. Noble, Morgan Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems*, 8(4), 1995.

[14] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases, Theory, Design, and Implementation.* Benjamin/Cummings Publisher Company, 1993.

[15] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Computer Review*, 26(2), April 1996.

[16] Treno Web Page. http://www.psc.edu/networking/treno_info.html.

[17] Upper Atmospheric Research Collaboratory (UARC) project homepage. http://www.si.umich.edu/UARC/.

[18] Tak W. Yan and Hector Garcia-Molina. SIFT - a tool for wide-area information dissemination. In *Proceedings of USENIX 1995 Technical Conference*, January 1995.

[19] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, September 1993.

Creating a Personal Web Notebook

Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu
http://glimpse.cs.arizona.edu/udi.html

ABSTRACT

This paper introduces a tool, called Nabbit, to go from the World-Wide Web to Your Own Web. Nabbit uses a copy-and-paste paradigm, adapted to the way the web is used, to provide a convenient personal notebook. While browsing, users can select, with the mouse, any part of an HTML page they are looking at, and Nabbit will copy that part with the original format — images, forms, links and all — to their own pages. The source, date, and even personal comments are copied as well. Collection of information becomes as simple as "here's what I want — click — I got it." Nabbit can be used to write reports interleaved with web content, maintain extended hot lists (with your own comments and even parts of pages), collect selected hits from search results, and much more.

1. Introduction

How do you remember something you have seen on the web? Besides writing the information down by hand, there are currently only two major methods for keeping track of web information: Adding the current page to a hot list (or bookmarks, or favorite list), or using the SaveAs command to save the contents of the page. Both methods work well for small number of pages, but they do not scale. Anyone who uses the web extensively is running against this problem.

Several methods have been suggested and employed. Our Warmlist tool [1] extends the hot-list concept by automatically saving the full text of all hot-list entries and providing search. There are tools that capture *everything* you load and allow you to search and browse the full history of your browsing. The original Mosaic had annotation features [2], which somehow did not catch on as much as they should have.

The main issue here is convenience. The web itself is so popular because of convenience — with a few clicks one can get the world. However, the process of collecting relevant information and putting it together cannot be fully automated. Saving files cannot be completely transparent, because what to save depends on the users. If you save too much you run into the same scale problems when you need to use that information. If you save too little, or if it is too cumbersome to save, you lose information.

We need tools that give users the power to decide what exactly to save and what to do with it, but let them do it so conveniently that it does not interfere with their normal browsing. This is exactly what Nabbit is designed to do.

[1] **Supported in part by NSF grant CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052.**

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

Nabbit works with two Netscape windows, one for browsing and one for collecting information; in other words, one for input and one for output. (Nabbit is implemented at the moment only for Netscape running on UNIX.) The output window can be iconized and out of view. Let's say that you want to collect information about a certain topic and you use a search engine. Not all the hits it gives you will be relevant. You would like to select some of them and remember only those. With Nabbit, all you do is select with the mouse the part of the page you want to remember, and click on the "Copy" button on Nabbit's window. You may select hits number 4, 6, and 7, then follow one of the links, and copy half of that page, go back to the search results, get another set of hits and select hits 14, and 18, follow several links to maybe a local search engine, copy that form (so you can perform searches on it later on), copy a table, a set of links, email addresses, and an interesting paragraph. Everything you copy is being assembled into one HTML page, which you can view on the output window. All copying is done simply by selecting the part you want with the mouse and clicking on Copy. Links to the original sources, dates, and optional mirror copies are automatically added. You can save (publish) that page at any time, you have unlimited "undo"s, you can (full text) search all the pages you collected, you can load an old page to the output (or input) windows and add to it, and so on.

Everything that Nabbit does can, of course, be done with other means. For example, a good HTML editor (such as the one that comes with Netscape 3 Gold, or IE 4.0) allows copying of parts of pages. But it is inherently more complex for the user, because it is not integrated with the browsing. The page needs to be saved into a file, the editor needs to be started, the required part needs to be copied to another place, and only then can the browsing proceed. The user is distracted enough not to do it on a regular basis. With Nabbit this whole process takes one click.

Examples of the use of Nabbit are given in the Appendix (and the reader may want to jump there early). We first describe the main algorithm behind the capabilities of Nabbit.

2. The Main Algorithm

The heart of Nabbit is a copy-and-paste paradigm. To move HTML code from one place to another, Nabbit requires only that you select what you see on the browser's window. Nabbit then takes the selection (which is always simple text) from the clipboard and figures out the appropriate HTML code for it. In a nutshell, it works as follows: In addition to the clipboard, Nabbit also fetches the HTML source of the current page (using Netscape's remote command facilities, described later). Given a text selection and a source HTML, Nabbit extracts the text from the HTML code, and then employs an *approximate string matching algorithm* to find where the selection best matches the text. Once the location of the selection is found in the HTML code, only the tags that are relevant to that selection are taken, forming a stand-alone HTML piece that corresponds to the original selection as it looked on the browser. This part is not easy, because (practical) HTML is not as clean as it looks. (Actually, HTML often doesn't even *look* clean.) The new HTML piece is then shown on the output window, added to whatever is currently present in the output window, or is used by Nabbit as a base for fetching more documents. That's the essence of the algorithm. Let's go into a few more details.

Let's call the text selection that is copied to the clipboard T, and the source of the HTML document S. Both T and S are strings of characters. We need to find the location in S that generated the text T. In general, all the characters in T appear somewhere in S, although there are a few exceptions. One obvious exception is white space, which may be generated by some HTML tags (like <P> or <BR>). Another, less obvious, exception is list numbering generated by the <OL> tag. These numbers will be copied to the clipboard, but of course they are not explicitly in S. (The bullets generated by <UL> are not copied to the clipboard, by the way.) There are also many examples of characters in S that are not in T. They include formatting commands, HTML tags, special characters, white space, and more.

To find the source of T in S, we first parse S to divide it into HTML tags and text. The general rule is that everything between the <> brackets are HTML tags and everything else is text. Again, there are exceptions. For example, the content of OPTION tags are outside the brackets but they do not appear as text in the browser and they cannot be copied. The TITLE tag is another example. We strip all white space, because the correlation it contributes is generally low. We then compare T to the text in S, but do so *approximately*. That is, we allow insertions and deletions both in the text and in T. The algorithm we chose is not a well-known one [3]; it allows to set arbitrary costs for each insertion and each deletion based not only on the characters but also on their location in both strings. (To be honest, another reason for this choice was that the code was immediately available to us.)

After the approximate string matching is performed and a location of T is found in S, we need to reconstruct the HTML formatting. Our first attempt was to perform a complete parsing of HTML and then to re-build the selection HTML from that. While this looks like the right approach theoretically, in practice it did not work well. We found that a large percentage of HTML pages on the web — even pages generated by ''authoring programs'' — contain major HTML errors. We could not afford to simply output ''HTML error'' (like compilers do) and quit. Instead of trying to fix those errors, we decided that the best solution is to leave them in! After all, when you copy something you would like it to appear in the same way. If the errors are left untouched, they will be handled on the copy in the same way they are handled in the source. This turned out to work very well.

Overall, the algorithm consists of 6 steps:

1. remove white space from both T and S,
2. divide the HTML into tags and text, and store the original positions of both,
3. find the location of T in the text part of S (allowing up to 50% insertions or deletions),
4. determine which tags have no effect on T and can be ignored,
5. put the relevant tags back in their original place around T,
6. add extra information (such as a link to the source, and date).

Since web pages are almost always pretty short, and the network is (still) almost always relatively slow, all these processing steps are negligible in terms of running times. (The pattern matching part of the algorithm, which is the most CPU intensive, is written in C for best performance.)

Although we do not try to understand the structure of a page, we do make some attempts to ensure that the resulting HTML code is good. Here are some important examples.

- We close all open tags. If a certain page contains an <A> tag (link) or a <B> tag (bold) that were not closed (not uncommon), we don't want them to affect everything after the copy.
- If the selection contains any part of a FORM, we copy the whole FORM. Partial FORMs are not always workable, and it's not always clear from looking at the browser where the FORM begins and where it ends.
- We do allow copying parts of tables. In that case, we try to make the partial table as close in formatting to the original table as we can. Unlike FORMs, partial tables, although they may look awkward, can be very useful.

3. The Interaction with the Browser

Nabbit communicates with Netscape through Netscape's remote control mechanism [4], a wonderful yet not widely known facility provided by Netscape only in its UNIX versions. (We are currently working on ports to other platforms, notably Windows and IE, which will require different communication mechanisms. We believe that the same approach will work, with the communication done through the browsers' APIs, although the code will be more complex.) The remote control mechanism allows activation of Netscape menu items on any Netscape window from another process. For example, one can save the source HTML of the current page to fileName by issuing

```
netscape -id id_number -remote
saveAs(fileName)
```

where id_number is the window id given to the netscape window by the X window manager. Similarly,

```
netscape -id id_number -remote
openFile(fileName)
```

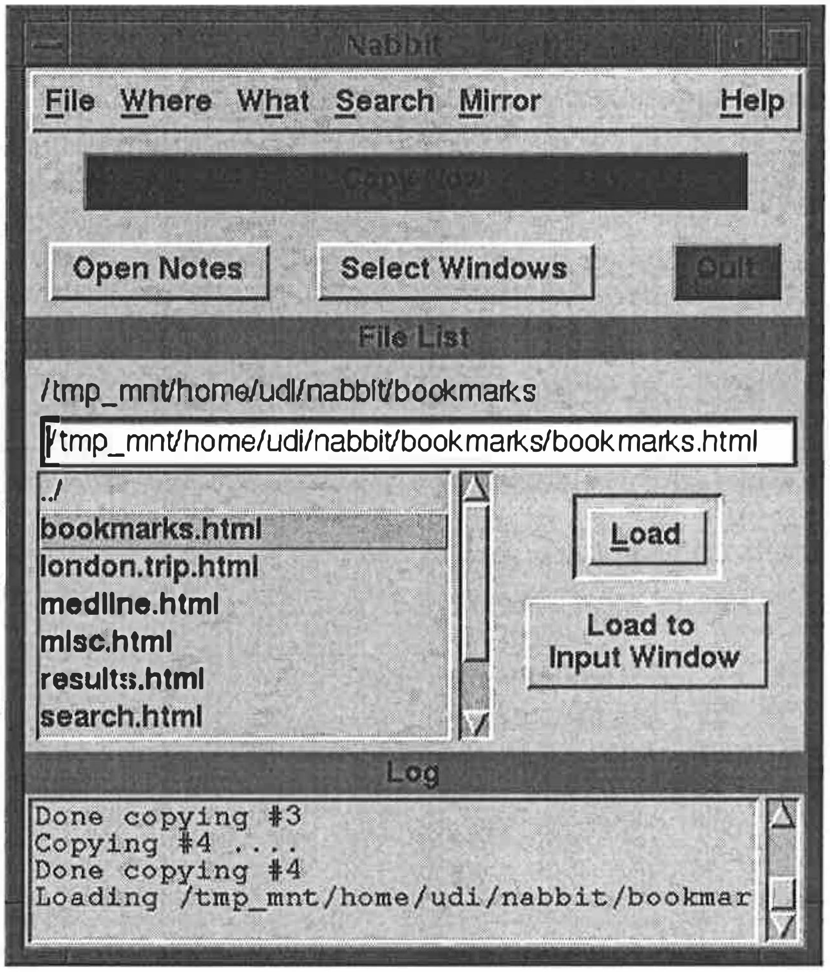
brings the contents of fileName into the netscape window. These are the only two actions we need. We use the saveAs mechanism in lieu of fetching the content based on a URL for two reasons. First, it is faster, because it usually copies the content directly from memory and there is no need to go again to the network. Second and just as important, it allows Nabbit to work on results of searches that used the POST action. Such results cannot be fetched from the URL. The communication between Nabbit and Netscape was mostly borrowed from NetShell [5].

When Nabbit is started, it obtains from the X window manager the list of all netscape windows (using the xwininfo program) and present them to the user to choose which one will be the input and output windows. The appropriate id_numbers are then used. This choice can be changed at any time. The X window manager is also used by Nabbit to obtain the URL of the current page. We found no easy way to get that information from Netscape (it is not available in any menu item). But Netscape does tell the X window manager the URL so it can be shown in its box. The URL is not essential to Nabbit's operations, but it is useful to include it as part of the copy, so users can go back later to the source.

Overall, even though the interprocess communication between Netscape, the X window manager, and Nabbit is currently mostly ad-hoc, it is extremely effective and easy to use.

4. The User Interface

The command window of Nabbit is shown in Figure 1 (it looks much better in color). The ''copy now'' button does most of the work. The ''File list'' area that occupies the middle part of the window shows the files that were saved before, and it allows to load any of them (or any other file) to either the output or input windows. The ''Where'' and ''What'' selection menus are shown in Figure 2. They give several options of what to copy and where to put it. The Notes button opens a text window into which the user can type (or copy) any notes they wish to add.



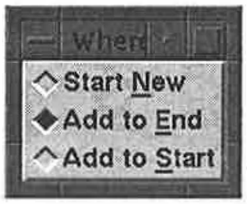
/tmp_mnt/home/udi/nabbit/bookmarks

/tmp_mnt/home/udi/nabbit/bookmarks/bookmarks.html

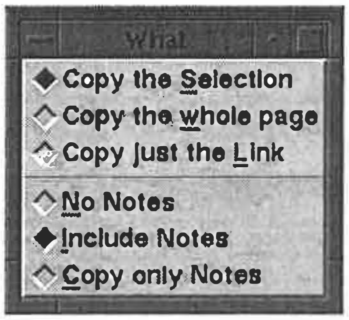
../
bookmarks.html
london.trip.html
medline.html
misc.html
results.html
search.html

```
Done copying #3
Copying #4 ....
Done copying #4
Loading /tmp_mnt/home/udi/nabbit/bookmar
```

Figure 1: Nabbit's user interface



◇ Start New
◆ Add to End
◇ Add to Start



◆ Copy the Selection
◇ Copy the whole page
◇ Copy just the Link
◇ No Notes
◆ Include Notes
◇ Copy only Notes

Figure 2: The Where and What options

Other menu items include Search — using glimpse — and Mirror, which mirrors into the local disk the current page and/or pages and images pointed from that page (only one page mirror is currently implemented).

Since every copy involves loading a different page to the browser, we get a very nice side effect of having unlimited undo's! To undo a copy, simply press ''back'' on the output window.

5. Conclusions and Further Work

Nabbit provides a convenient and natural way to take notes while browsing the web. As the web is becoming the primary interface to information, it is essential to find better ways to capture that information. Besides just copying parts of regular web pages, Nabbit can be effectively used to collect results of database searches, which will be more and more important as more databases are connected to the web. Our next development step is to extend the publishing capabilities of Nabbit, allowing people to easily publish their ''notes'' in their organizations, intranets, or throughout the web. This will provide another way to collaborate and use the web more effectively.

6. Acknowledgements

David Lipman and Jim Ostell of the National Center of Biotechnology Information at NIH triggered this work by suggesting to me the problem and its applications, and contributed many useful comments.

References

[1] P. Klark, and U. Manber, ''Developing a Personal Internet Assistant,'' *Proceedings of ED-Media 95, World Conf. on Multimedia and Hypermedia*, Graz, Austria (June 1995), pp. 372–377.

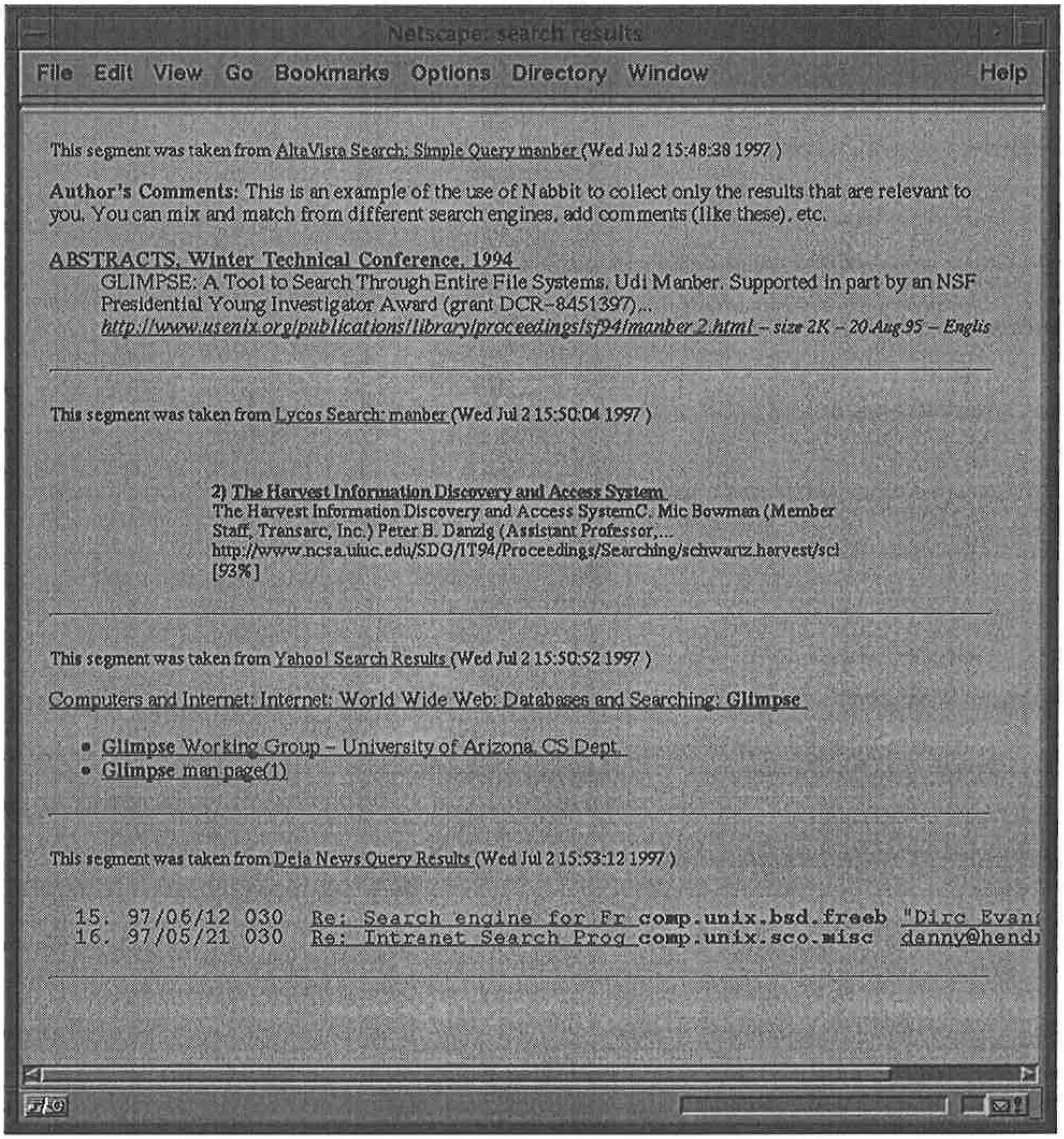
[2] Mosaic User's Guide: Annotations, http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-on-annotate-win.html

[3] U. Manber and S. Wu, ''Approximate String Matching With Arbitrary Costs for Text and Hypertext,'' *Proc. of the IAPR International Workshop on Structural and Syntactic Pattern Recognition*, Bern, Switzerland (August 1992), pp. 22–33.

[4] Zawinski, J., Remote Control of UNIX Netscape, http://home.netscape.com/newsref/std/x-remote.html (December 1994).

[5] D. Zhang, and Udi Manber, ''NetShell — Customized Handling of WEB Information,'' http://www.cs.arizona.edu/netshell (June 1996).

Appendix: Examples



This segment was taken from AltaVista Search: Simple Query manber (Wed Jul 2 15:48:38 1997)

Author's Comments: This is an example of the use of Nabbit to collect only the results that are relevant to you. You can mix and match from different search engines, add comments (like these), etc.

ABSTRACTS, Winter Technical Conference, 1994
GLIMPSE: A Tool to Search Through Entire File Systems. Udi Manber. Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397)...
http://www.usenix.org/publications/library/proceedings/sf94/manber 2.html – size 2K – 20 Aug 95 – Englis

This segment was taken from Lycos Search: manber (Wed Jul 2 15:50:04 1997)

2) The Harvest Information Discovery and Access System
The Harvest Information Discovery and Access SystemC. Mic Bowman (Member Staff, Transarc, Inc.) Peter B. Danzig (Assistant Professor,...
http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/schwartz.harvest/sci
[93%]

This segment was taken from Yahoo! Search Results (Wed Jul 2 15:50:52 1997)

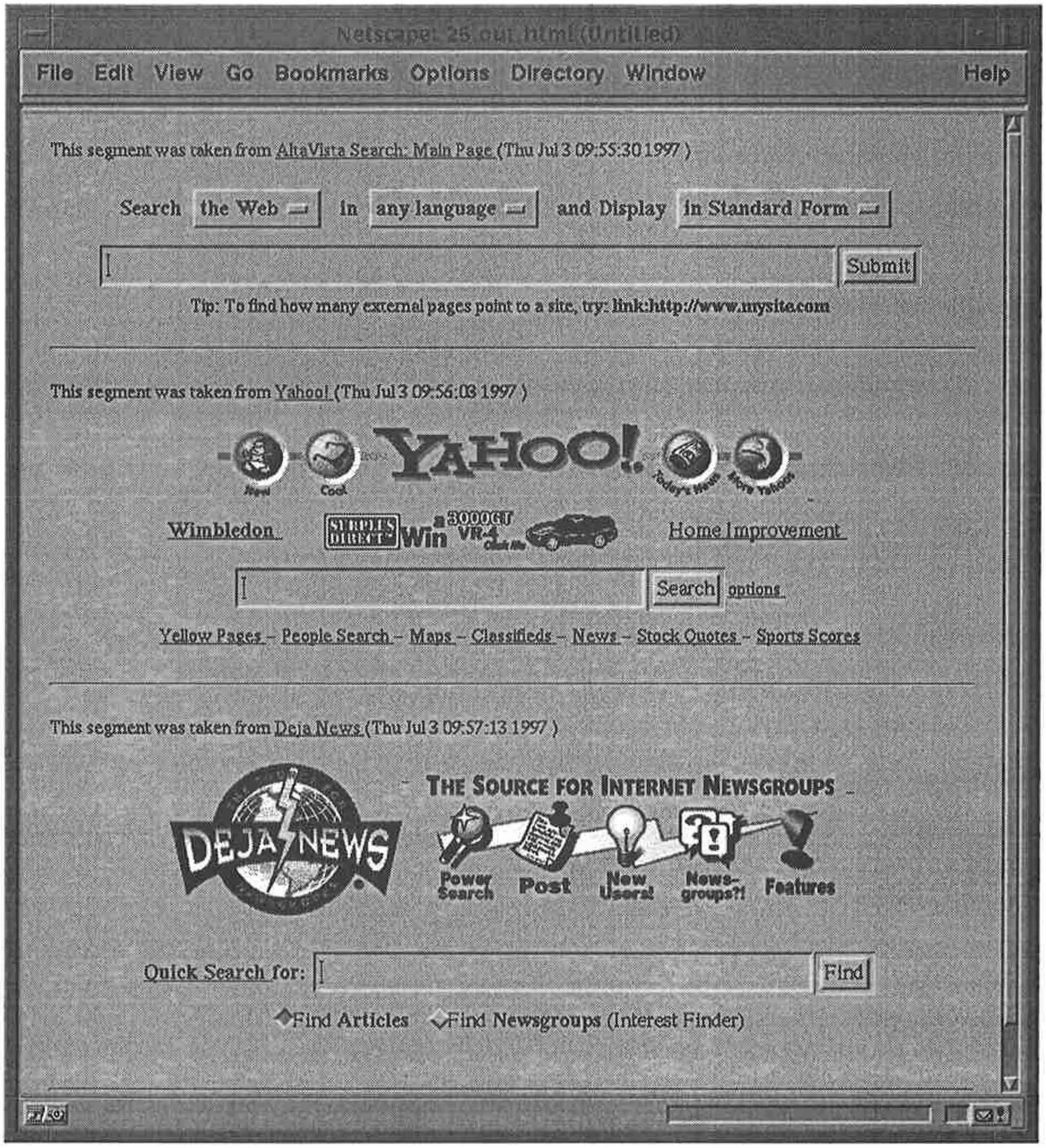
Computers and Internet: Internet: World Wide Web: Databases and Searching: **Glimpse**

- **Glimpse** Working Group – University of Arizona, CS Dept.
- **Glimpse** man page(1)

This segment was taken from Deja News Query Results (Wed Jul 2 15:53:12 1997)

```
 15. 97/06/12 030  Re: Search engine for Fr comp.unix.bsd.freeb "Dirc Evan
 16. 97/05/21 030  Re: Intranet Search Prog comp.unix.sco.misc  danny@hend
```

Figure 3: Examples of collecting Search Results


Netscape: 25.out.html (Untitled)
File Edit View Go Bookmarks Options Directory Window Help
Submit
YAHOO!
New
Cool
Today's News
More Yahoos
Wimbledon
SURPLUS DIRECT Win a 3000GT VR-4 Click Me
Home Improvement
Search
options
Yellow Pages – People Search – Maps – Classifieds – News – Stock Quotes – Sports Scores
DEJA NEWS
THE SOURCE FOR INTERNET NEWSGROUPS
Power Search
Post
New Users!
News-groups?!
Features
Find

This segment was taken from AltaVista Search: Main Page (Thu Jul 3 09:55:30 1997)

Search the Web in any language and Display in Standard Form

Tip: To find how many external pages point to a site, try: **link:http://www.mysite.com**

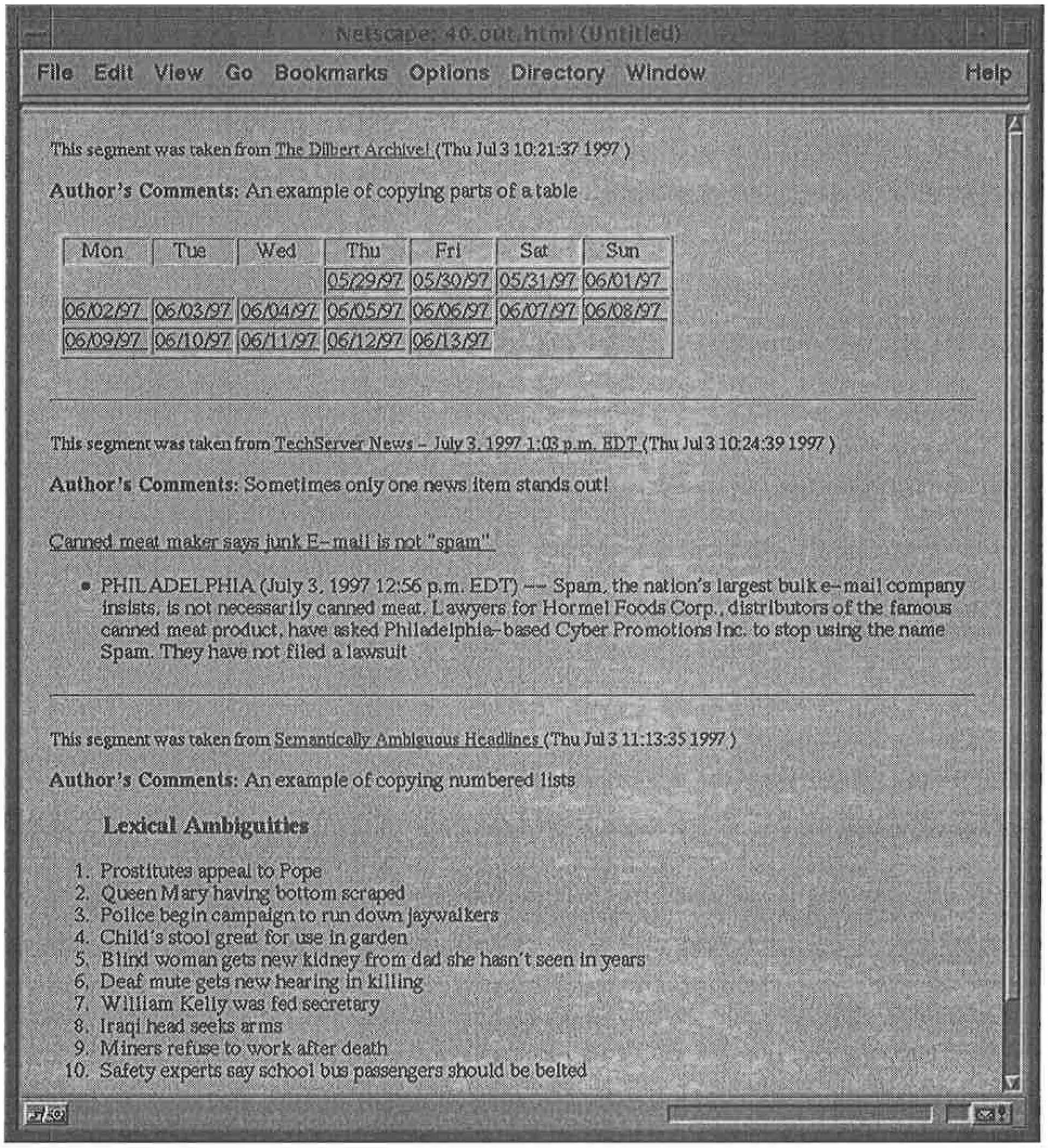
This segment was taken from Yahoo! (Thu Jul 3 09:56:03 1997)

This segment was taken from Deja News (Thu Jul 3 09:57:13 1997)

Quick Search for:

◆Find Articles ◇Find Newsgroups (Interest Finder)

Figure 4: Examples of collecting Search Forms


Netscape: 40.out.html (Untitled)
File Edit View Go Bookmarks Options Directory Window Help

This segment was taken from The Dilbert Archive! (Thu Jul 3 10:21:37 1997)

Author's Comments: An example of copying parts of a table

Mon	Tue	Wed	Thu	Fri	Sat	Sun
			05/29/97	05/30/97	05/31/97	06/01/97
06/02/97	06/03/97	06/04/97	06/05/97	06/06/97	06/07/97	06/08/97
06/09/97	06/10/97	06/11/97	06/12/97	06/13/97		

This segment was taken from TechServer News - July 3, 1997 1:03 p.m. EDT (Thu Jul 3 10:24:39 1997)

Author's Comments: Sometimes only one news item stands out!

Canned meat maker says junk E-mail is not "spam"

- PHILADELPHIA (July 3, 1997 12:56 p.m. EDT) -- Spam, the nation's largest bulk e-mail company insists, is not necessarily canned meat. Lawyers for Hormel Foods Corp., distributors of the famous canned meat product, have asked Philadelphia-based Cyber Promotions Inc. to stop using the name Spam. They have not filed a lawsuit

This segment was taken from Semantically Ambiguous Headlines (Thu Jul 3 11:13:35 1997)

Author's Comments: An example of copying numbered lists

Lexical Ambiguities

1. Prostitutes appeal to Pope
2. Queen Mary having bottom scraped
3. Police begin campaign to run down jaywalkers
4. Child's stool great for use in garden
5. Blind woman gets new kidney from dad she hasn't seen in years
6. Deaf mute gets new hearing in killing
7. William Kelly was fed secretary
8. Iraqi head seeks arms
9. Miners refuse to work after death
10. Safety experts say school bus passengers should be belted

Figure 5: Miscellaneous Examples

Cost-Aware WWW Proxy Caching Algorithms

Pei Cao
Department of Computer Science,
University of Wisconsin-Madison.
cao@cs.wisc.edu

Sandy Irani
Information and Computer Science Department,
University of California-Irvine.
irani@ics.uci.edu

Abstract

Web caches can not only reduce network traffic and downloading latency, but can also affect the distribution of web traffic over the network through cost-aware caching. This paper introduces GreedyDual-Size, which incorporates locality with cost and size concerns in a simple and non-parameterized fashion for high performance. Trace-driven simulations show that with the appropriate cost definition, GreedyDual-Size outperforms existing web cache replacement algorithms in many aspects, including hit ratios, latency reduction and network cost reduction. In addition, GreedyDual-Size can potentially improve the performance of main-memory caching of Web documents.

1 Introduction

As the World Wide Web has grown in popularity in recent years, the percentage of network traffic due to HTTP requests has steadily increased. Recent reports show that Web traffic has constituted 40% of the network traffic in 1996, compared to only 19% in 1994. Since the majority of Web documents requested are static documents (i.e. home pages, audio and video files), caching at various network points provides a natural way to reduce web traffic. A common form of web caching is caching at HTTP proxies, which are intermediateries between browser processes and web servers on the Internet (for example, one can choose a proxy by setting the network preference in the Netscape Navigator[1]).

There are many benefits of proxy caching. It reduces network traffic, average latency of fetching Web documents, and the load on busy Web servers. Since documents are stored at the proxy cache, many HTTP requests can be satisfied directly from the cache instead of generating traffic to and from the Web server. Numerous studies [WASAF96] have shown that the hit ratio for Web proxy caches can be as high as over 50%. This means that if proxy caching is utilized extensively, the network traffic can be reduced significantly.

Key to the effectiveness of proxy caches is a document replacement algorithm that can yield high hit ratio. Unfortunately, techniques developed for file caching and virtual memory page replacement do not necessarily transfer to Web caching.

There are three primary differences between Web caching and conventional paging problems. First, web caching is variable-size caching: due to the restriction in HTTP protocols that support whole file transfers only, a cache hit only happens if the entire file is cached, and web documents vary dramatically in size depending on the information they carry (text, image, video, etc.). Second, web pages take different amounts of time to download, even if they are of the same size. A proxy that wishes to reduce the average latency of web accesses may want to adjust its replacement strategy based on the download latency. Third, access streams seen by the proxy cache are the union of web access streams from tens to thousands of users, instead of coming from a few programmed sources as in the case of virtual memory paging.

Proxy caches are in a unique position to affect web traffic on the Internet. Since the replacement algorithm decides which documents are cached and which documents are replaced, it affects which future requests will be cache hits. Thus, if the institution employing the proxy must pay more on some network links than others, the replacement algorithm can favor expensive documents (i.e. those travelling through the expensive links) over cheap documents. If it is known that certain network paths are heavily congested, the caching algorithm can retain more documents which must travel on congested paths. The proxy cache can reduce its contribution to the network router load by preferentially caching documents that travel more hops. Web cache replace-

[1]Navigator is a trademark of Netscape Inc.

ment algorithms can incorporate these considerations by associating an appropriate *network cost* with every document, and minimizing the total cost incurred over a particular access stream.

Today, most proxy systems use some form of the Least-Recently-Used replacement algorithm. Though some proxy systems also consider the time-to-live fields of the documents and replace expired documents first, studies have found that time-to-live fields rarely correspond exactly to the actual life time of the document and it is better to keep expired-but-recently-used documents in the cache and validate them by querying the server [LC97]. The advantage of LRU is its simplicity; the disadvantage is that it does not take into account file sizes or latency and might not give the best hit ratio.

Many Web caching algorithms have been proposed to address the size and latency concerns. We are aware of at least nine algorithms, from the simple to the very elaborate, proposed and evaluated in separate papers, some of which give conflicting conclusions. This naturally leads to a state of confusion over which algorithm should be used. In addition, none of the existing algorithms address the network cost concerns.

In this paper, we introduce a new algorithm, called GreedyDual-Size, which combines locality, size and latency/cost concerns effectively to achieve the best overall performance. GreedyDual-Size is a variation on a simple and elegant algorithm called GreedyDual [You91b], which handles uniform-size variable-cost cache replacement. Using trace-driven simulation, we show that GreedyDual-Size with appropriate cost definitions out-performs the various "champion" web caching algorithms in existing studies on a number of performance issues, including hit ratios, latency reduction, and network cost reduction.

2 Existing Results

The *size* and *cost* concerns make web caching a much more complicated problem than traditional caching. Below we first summarize the existing theoretical results, then take a look at a variety of web caching algorithms proposed so far.

2.1 Existing Theoretical Results

There are a number of results on the optimal offline replacement algorithms and online competitive algorithms on simplified versions of the Web caching problem.

The variable document sizes in web caching make it much more complicated to determine an optimal offline replacement algorithm. If one is given a sequence of requests to uniform size blocks of memory, it is well known that the simple rule of evicting the block whose next request is farthest in the future will yield the optimal performance [Bel66]. In the variable-size case, no such offline algorithm is known. In fact, it is known that determining the optimal performance is NP-hard [Ho97], although there is an algorithm which can approximate the optimal to within a logarithmic factor [Ir97]. The approximation factor is logarithmic in the maximum number of bytes that can fit in the cache, which we will call k.

For the cost consideration, there have been several algorithms developed for the uniform-size variable-cost paging problem. GreedyDual [You91b], is actually a range of algorithms which include a generalization of LRU and a generalization of FIFO. The name GreedyDual comes from the technique used to prove that this entire range of algorithms is optimal according to its *competitive ratio*. The competitive ratio is essentially the maximum ratio of the algorithms cost to the optimal offline algorithm's cost over all possible request sequences. (For an introduction to *competitive analysis*, see [ST85]).

We have generalized the result in [You91b] to show that our algorithm GreedyDual-Size, which handles documents of differing sizes and differing cost (described in Section 4), also has an optimal competitive ratio. Interestingly, it is also known that LRU has an optimal competitive ratio when the page size can vary and the cost of fetching a document is the same for all documents or proportional to the size of a document [FKIP96].

2.2 Existing Document Replacement Algorithms

We describe nine cache replacement algorithms proposed in recent studies, which attempt to minimize various cost metrics, such as miss ratio, byte miss ratio, average latency, and total cost. Below we give a brief description of all of them. In describing the various algorithms, it is convenient to view each request for a document as being satisfied in the following way: the algorithm brings the newly requested document into the cache and then evicts documents until the capacity of the cache is no longer exceeded. Algorithms are then distinguished by how they choose which documents to evict. This view allows for the possibility that the requested document itself may be evicted upon its arrival into the cache, which means it replaces no other document in the cache.

- **Least-Recently-Used** (LRU) evicts the document which was requested the least recently.

- **Least-Frequently-Used** (LFU) evicts the document which is accessed least frequently.
- **Size** [WASAF96] evicts the largest document.
- **LRU-Threshold** [ASAWF95] is the same as LRU, except documents larger than a certain threshold size are never cached;
- **Log(Size)+LRU** [ASAWF95] evicts the document who has the largest log(size) and is the least recently used document among all documents with the same log(size).
- **Hyper-G** [WASAF96] is a refinement of LFU with last access time and size considerations;
- **Pitkow/Recker** [WASAF96] removes the least-recently-used document, except if all documents are accessed today, in which case the largest one is removed;
- **Lowest-Latency-First** [WA97] tries to minimize average latency by removing the document with the lowest download latency first;
- **Hybrid**, introduced in [WA97], is aimed at reducing the total latency. A function is computed for each document which is designed to capture the utility of retaining a given document in the cache. The document with the smallest function value is then evicted. The function for a document p located at server s depends on the following parameters: c_s, the time to connect with server s, b_s the bandwidth to server s, n_p the number of times p has been requested since it was brought into the cache, and z_p, the size (in bytes) of document p. The function is defined as:

  $$\frac{\left(c_s + \frac{W_b}{b_s}\right)(n_p)^{W_n}}{z_p}$$

  where W_b and W_n are constants. Estimates for c_s and b_s are based on the the times to fetch documents from server s in the recent past.
- **Lowest Relative Value** (LRV), introduced in [LRV97], includes the cost and size of a document in the calculation of a value that estimates the utility of keeping a document in the cache. The algorithm evicts the document with the lowest value. The calculation of the value is based on extensive empirical analysis of trace data. For a given i, let P_i denote the probability that a document is requested $i+1$ times given that it is requested i times. P_i is estimated in an online manner by taking the ratio D_{i+1}/D_i, where D_i is the total number of documents seen so far which have been requested at least i times in the trace. $P_i(s)$ is the same as P_i except the value is determined by restricting the count only to pages of size s. Furthermore, let $1-D(t)$ be the probability that a page is requested again as a function of the time (in seconds) since its last request t; $D(t)$ is estimated as

  $$D(t) = .035\log(t+1) + .45\left(1 - e^{\frac{-t}{2e6}}\right).$$

  Then for a particular document d of size s and cost c, if the last request to d is the i'th request to it, and the last request was made t seconds ago, d's value in LRV is calculated as:

  $$V(i,t,s) = \begin{cases} P_1(s)(1-D(t)) * c/s & \text{if } i = 1 \\ P_i(1-D(t)) * c/s & \text{otherwise} \end{cases}$$

  Among all documents, LRV evict the one with the lowest value. Thus, LRV takes into account locality, cost and size of a document.

Existing studies using actual Web proxy traces narrowed down the choice for proxy replacement algorithms to LRU, SIZE, Hybrid and LRV. Results in [WASAF96, ASAWF95] show that SIZE performs better than LFU, LRU-threshold, Log(size)+LRU, Hyper-G and Pitkow/Recker. Results in [WASAF96] also show that SIZE outperforms LRU in most situations. However, a different study [LRV97] shows that LRU outperforms SIZE in terms of byte hit rate. Comparing LFU and LRU, our experiments show that though LFU can outperform LRU slightly when the cache size is very small, in most cases LFU performs worse than LRU. In terms of minimizing latency, [WA97] show that Hybrid performs better than Lowest-Latency-First. Finally, [LRV97] shows that LRV outperforms both LRU and SIZE in terms of hit ratio and byte hit ratio. One disadvantage of both Hybrid and LRV is their heavy parameterization, which leaves one uncertain about their performance across access streams.

However, the studies offer no conclusion on which algorithm a proxy should use. Essentially, the problem is finding an algorithm that can combine the observed access pattern with the cost and size considerations.

2.2.1 Implementation Concerns

The above "champion" algorithms vary in time and space complexity. In the cases when there are a large number of documents in the cache, this can have a dramatic effect on the time required to determine which document to evict.

- LRU can be implement easily with $O(1)$ overhead per cached file and $O(1)$ time per access;
- Size can be implemented by maintaining a priority queue on the documents in memory based on their size. Since the size of a document does not change, handling a hit requires $O(1)$ time and handling an eviction requires $O(\log k)$ time, where k is the number of cached documents.
- Hybrid is also implemented using a priority queue, thus requiring $O(\log k)$ time to find a replacement. Furthermore, it requires an array keeping track of the average latency and bandwidth for every Web server. It is used in estimating the downloading latency of a web page. This requires extra storage. In addition, since the estimate is updated every time a connection to the server is made, a faithful implementation requires updating many pages' latency estimation. We found this prohibitively time-consuming, and we omit the step in the implementation. We find that omitting the step does not affect our results significantly.
- LRV requires $O(1)$ storage per cached file plus some bookkeeping information. If the *Cost* in LRV is proportional to *Size*, the authors of the algorithm suggests an efficient method that can find the replacement in $O(1)$ time, though the constants can be large. If *Cost* is arbitrary, then $O(k)$ time is needed to find a replacement. We also found that the cost of calculating $D(t)$ are very high, since it uses *log* and *exp*.

Another concern about both Hybrid and LRV is that they employ constants which might have to be tuned to the patterns in the request stream. For Hybrid, we use the values which were used in [WA97] in our simulations. We did not experiment with tuning those constants to improve the performance of Hybrid.

Though LRV can incorporate arbitrary network costs associated with documents, the $O(k)$ computational complexity of finding a replacement can be prohibitively expensive. The problem is that $D(t)$ has to be recalculated for every document every time some document has to be replaced. The overhead makes LRV impractical for proxy caches that wish to take network costs into consideration.

3 Web Proxy Traces

As the conclusions from a trace-driven study inevitably depend on the traces, we tried to gather as many traces as possible. We were successful in obtaining the following traces of HTTP requests going through Web proxies:

- Digital Equipment Corporation Web Proxy server traces [DEC96](Aug-Sep 1996), servicing about 17,000 workstations, for a period of 25 days, containing a total of about 24,000,000 accesses;
- University of Virginia proxy server and client traces [WASAF96] (Feb-Oct 1995), containing four sets of traces, each servicing from 25 to 61 workstations, containing from 13,127 to 227,210 accesses;
- Boston University client traces [CBC95](Nov 1994 - May 1995), containing two sets of traces, one servicing 5 workstations (17,008 accesses), the other 32 workstations (118,105 accesses);

We are in the process of obtaining more traces from other sources.

We present the results of fourteen traces. They include all of Virginia Tech and Boston University traces, and eight subsets of the DEC traces. The subsets are Web accesses made by users 0-512, and users 1024-2048, in each week, for the three and a half weeks period from Aug. 29 to Sep. 22, 1996. The use of the subsets is partly due to our current simulator's limitation (it cannot simulate more than two million requests at a time), and partly due to our observation that a *caching* proxy server built out of a high-end workstation can only service about 512 users at a time.

We perform some necessary pre-processing over the traces. For the DEC traces, we simulated only those requests whose replies are cacheable as specified in HTTP 1.1 [HT97] (i.e. GET or HEAD requests with status 200, 203, 206, 300, or 301, and not a "cgi_bin" request). In addition, we do not include those requests that are queries (i.e. "?" appears in the URL), though such requests are a small fraction of total cacheable requests (around 3% to 5%). For Virginia Tech traces, we simulated only the "GET" requests with reply status 200 and a known reply size. Thus, our numbers differ from what are reported in [WASAF96]. The Virginia Tech traces unfortunately do not come with latency information. For Boston University traces, we simulated only those requests that are not serviced out of browser caches.

3.1 Locality in Web Accesses

In the search for an effective replacement algorithm, we analyzed the traces to understand the access patterns of Web requests seen by the proxies. The strik-

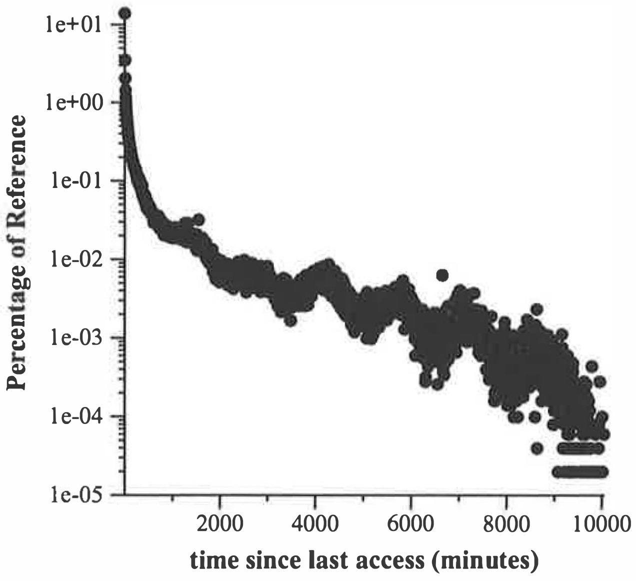


Figure 1: Percentage of references to documents whose last accesses are t minutes ago, for t from 5 to 10000.

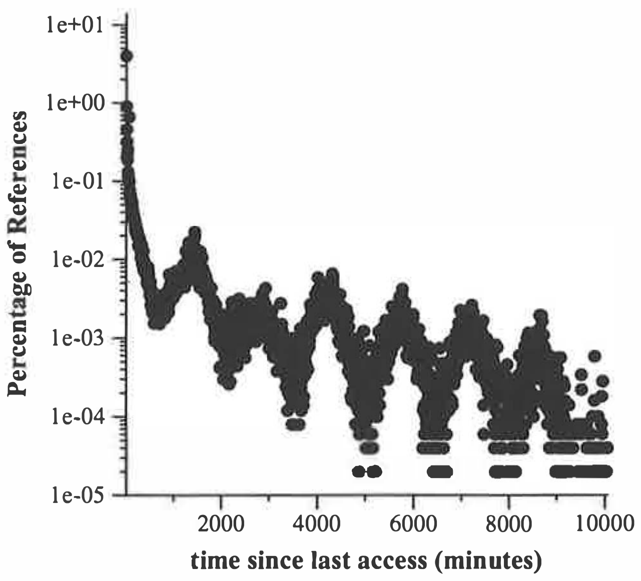


Figure 2: Percentage of references as a function of time since last access *by the same user*.

ing property we found is that all traces exhibit excellent long-term locality.

Figure 1 shows the percentage of references to a document whose last reference is t minutes ago, for t from 5 to 10000, in the DEC traces for the period from Sep. 12 to Sep. 18. In other words, the figure shows the probability of a document being accessed again as a function of the time since the last access to this document. The graphs for other traces are similar to the one shown here. Clearly, the probability of reference drops significantly as the time since last reference increases (note that the y-axis is in logarithmic scale), with occasional spikes around multiples of 24 hours.

Figure 3 shows the *accumulative* percentage of references to documents whose last references are less than t minutes ago, for the entire DEC traces from Aug. 29 to Sep. 22. The dashed curve on the graph shows the corresponding percentage of bytes referenced. In Figure 3, which uses linear scale for the y-axis, and logarithmic scale for the x-axis, we see that the curves are nearly linear. That is, the probability of a document being referenced again within t minutes is proportionally to $log(t)$, indicating that the probability of re-reference to documents referenced exactly t minutes ago can be modeled as k/t, where k is a constant.

A different study [LRV97] reached very similar conclusions on a different set of traces. Indeed, it is this observation that promoted the design of the function $D(t)$ in LRV. Since the studies find similar temporal locality patterns in the Web access traces, the probability density function of k/t has been used to simulate temporal locality behavior in a recent Web proxy benchmark [WPB].

There are two reasons for the good locality in Web accesses seen by the proxy. One is that each user's accesses tend to exhibit locality — figure 2 shows the probability that a document is accessed by a user t minutes after the last access by the same user, for DEC traces in the period from Sep. 12 to Sep. 18 (again, the figures for other traces are similar). Clearly, each user tends to re-access recently-read documents, and re-access documents that are read on a daily basis (note the spikes around 24 hours, 48 hours, etc. in the figure). Though one might expect that browsers' caches absorb the locality among the same user's accesses seen by the proxy, the results seems to indicate that this is not necessarily the case, and users are using proxy caches as an extension to the browser cache. [LRV97] observes the same phenomenon.

The other reason is that users' interests overlap in time — comparing figures 2 and 1, we can see that for the same t, the percentage in figure 1 is higher than that in figure 2. This indicates that part of the locality observed by the proxy comes from the fact that the proxy sees a merged stream of accesses from many independent users, who share a certain amount of common interests. Thus, we believe the locality observed is not particular to the traces described here, but rather a common characteristic of accesses seen by proxies with a large enough user community.

To further demonstrate the effect of inter-user sharing on hit ratios, Figure 4 shows the hit ratio

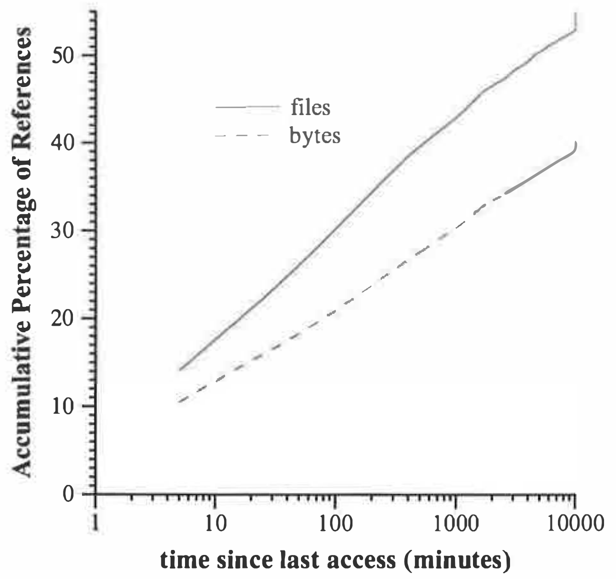


Figure 3: Percentage of references and percentage of bytes referenced to documents accessed less than t minutes ago (the accumulative version of Figure 1). *Note that the y-axis is in linear scale and the x-axis is in log scale.*

and byte hit ratio as a function of the size of the user group sharing the cache. The figures are quartile graphs [Tufte], the middle curve showing the median of the hit ratios of individual groups of clients in the DEC traces, and the other four points for each group size showing the minimum, the 25% percentile, the 75% percentile, and the maximum of the hit ratios of individual groups. The median hit ratios show an almost linear increase as the group size doubles.

In the absence of cost and size concerns, LRU is the optimal online algorithm for reference streams exhibiting good locality [CD73] (strictly speaking, those conforming to the LRU-stack model). However, in the Web context, replacing a more recently used but large file can yield a higher hit ratio than replacing a less recently used but small file. Similarly, replacing a more recently used but inexpensive file may yield a lower total cost than replacing a less recently used but expensive file. Thus, we need an algorithm that combines locality, size and cost considerations in a simple, online way that does not require tuning paramters according to the particular traces, and yet maximizes the overall performance.

4 GreedyDual-Size Algorithm

The original GreedyDual algorithm is proposed by Young [You91b]. It is actually a range of algorithms, but we focus one particular version which is a generalization of LRU. It is concerned with the case when pages in a cache have the same size, but incur different costs to fetch from a secondary storage. The algorithm associates a value, H, with each cached page p. Initially, when a page is brought into cache, H is set to be the cost of bringing the page into the cache (the cost is always non-negative). When a replacement needs to be made, the page with the lowest H value, min_H, is replaced, and then all pages reduce their H values by min_H. If a page is accessed, its H value is restored to the cost of bringing it into the cache. Thus, the H values of recently accessed pages retain a larger portion of the original cost than those of pages that have not been accessed for a long time. By reducing the H values as time goes on and restoring them upon access, the algorithm integrates the locality and cost concerns in a seamless fashion.

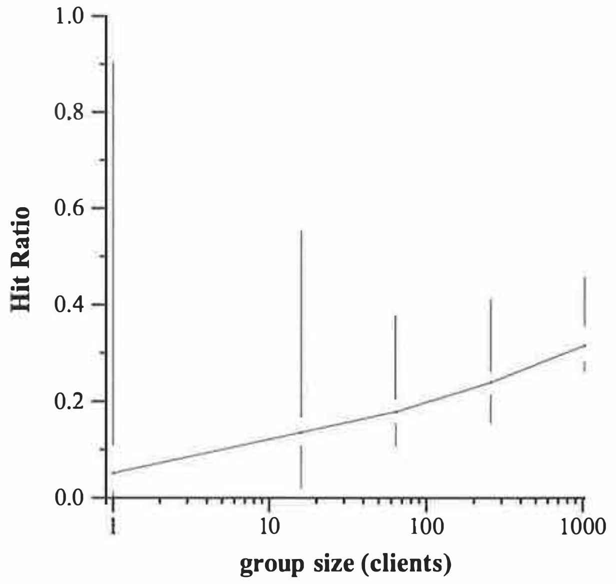
To incorporate the difference sizes of the document, we extend the GreedyDual algorithm by setting H to $cost/size$ upon an access to a document, where *cost* is the cost of bringing the document, and *size* is the size of the document in bytes. We called the extended version the GreedyDual-Size algorithm. The definition of *cost* depends on the goal of the replacement algorithm: *cost* is set to 1 if the goal is to maximize hit ratio, it is set to the downloading latency if the goal is to minimize average latency, and it is set to the network cost if the goal is to minimize the total cost.

At the first glance, GreedyDual-Size would require k subtractions when a replacement is made, where k is the number of documents in cache. However, a different way of recording H removes these subtractions. The idea is to keep an "inflation" value L, and let all future setting of H be offset by L. Figure 5 shows an efficient implementation of the algorithm.

Using this technique, GreedyDual-Size can be implemented by maintaining a priority queue on the documents, based on their H value. Handling a hit requires $O(\log k)$ time and handling an eviction requires $O(\log k)$ time, since in both cases a single item in the queue must be updated. More efficient implementations can be designed that make the common case of handling a hit requiring only $O(1)$ time.

Online-Optimality of GreedyDual-Size

It can be proven that GreedyDual-Size is online-optimal. For any sequence of accesses to documents with arbitrary sizes and arbitrary costs, the cost of cache misses under GreedyDual-Size is at most k times that under the offline optimal replacement algorithm, where k is the ratio of the cache size to the size of the smallest page. The ratio is the lowest achiev-



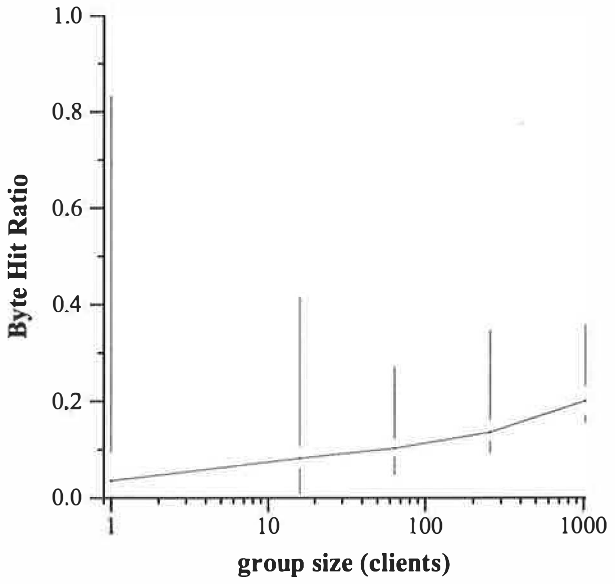


Figure 4: Hit ratio and byte hit ratio as a function of the size of the user group sharing the cache. The x-axis is in log scale.

```
Algorithm GREEDYDUAL:
Initialize $L \leftarrow 0$.
Process each request document in turn:
The current request is for document $p$:
(1) if $p$ is already in memory,
(2)      $H(p) \leftarrow L + c(p)/s(p)$.
(3) if $p$ is not in memory,
(4)      while there is not enough room
         in memory for $p$,
(5)           Let $L \leftarrow \min_{q \in M} H(q)$.
(6)           Evict $q$ such that $H(q) = L$.
(7)      Bring $p$ into memory and set
         $H(p) \leftarrow L + c(p)/s(p)$
    end
```

Figure 5: GreedyDual Algorithm.

able by any online replacement algorithm. Below is a proof of the online-optimality of GreedyDual-Size.

Neal Young proved in [You91b] that Greedy Dual for pages of uniform size is k-competitive. We prove here that the version which handles pages of multiple size is also k-competitive. (In both cases, k is defined to be the ratio of the size of the cache to the size of the smallest page). The proof below is based on a proof that another algorithm called BALANCE which also solves the multi-cost uniform-size paging problem is k-competitive [CKPV91].

All of the above bounds are tight, since we can always assume that all pages are as small as possible and have the same cost and invoke the lower bound of k on the competitive ratio for the uniform-size uniform-cost paging problem found in [ST85].

It should also be noted that the same bound can be proven for the version of the algorithm which uses $c(p)$ instead of $c(p)/s(p)$ in the description of the algorithm in Figure 5. Young has independently proven a generalization of the result below [You97]. The generalization covers the whole range of algorithms described in his original paper [You91b] instead of the particular version covered here.

Theorem 1 *GreedyDual-Size is k-competitive, where k is the ratio of the size of the cache to the size of the smallest document.*

Proof. We will charge each algorithm for the documents they evict instead of the documents they bring in. The difference between the two cost measures is at most an additive constant.

Each request for a document happens in a series of steps. First the optimal algorithm serves the request. Then each of the steps of GreedyDual-Size is executed in a separate step. Each step of each request happens at a different point in time.

It is straightforward to show by induction on time that for every document q in the cache

$$L \leq \min_{p \in M} H(p) \leq H(q) \leq L + \frac{c(q)}{s(q)}.$$

Let L_{final} be the last value of L. Let s_{min} denote the size of the smallest document. Let s_{cache} be the total size of the cache. Note that $k = s_{cache}/s_{min}$. We will first prove that the total cost of all documents which OPT evicts is at least $s_{min} \cdot L_{final}$. Then we will show that the total cost of all documents evicted by GreedyDual-Size is at most $s_{cache} \cdot L_{final}$.

Every time L increases, there is some document which GreedyDual-Size has in the cache which the optimal does not have in the cache. This is because L only increases when GreedyDual-Size has exceeded the size of its cache and must evict a document. Since the optimal algorithm has already satisfied the request, it has the requested document in the cache. Since the newly requested document does not fit in GreedyDual-Size's cache, GreedyDual-Size must have some document in the cache which the optimal does not have in the cache.

Consider a period of time in which GreedyDual-Size has p in its cache and the optimal does not. Such a period begins with the optimal evicting p from the cache and ends when either we evict p from the cache or the optimal brings p back in to the cache. We will attribute any increase in L which occurs during this period to the cost the optimal incurred in evicting p at the beginning of the period. The cost of evicting p is $c(p)$. The only thing we have to prove in establishing that the optimal cost is at least $s_{min} \cdot L_{final}$ is that L increases by at most $c(p)/s(p) \leq c(p)/s_{min}$ during the period.

Let L_1 be the value of L when the period begins. We know that at this time $H(p) \leq L_1 + c(p)/s(p)$. Furthermore, $H(p)$ does not change during this period. This is because $H(p)$ only increases when p is requested. p can only be requested on the last request of the period (because the period is defined to the period of time in which GreedyDual-Size has p in its cache and the optimal does not). If the last request of the period is to document p, then the optimal brings p into its cache, and hence the period ends before $H(p)$ increases. If the period ends by p's eviction, $H(p)$ remains the same until p is evicted. Since $H(p)$ is an upper bound for L, L can not increase to more than $L_1 + c(p)/s(p)$ during the entire period.

Now we must establish that the total cost of all documents evicted by GreedyDual-Size is at most $s_{cache} \cdot L_{final}$. Consider an eviction that GreedyDual-Size performs. Let p be the document that is evicted and let L_1 and L_2 be the values for L when it is brought in and evicted from the cache, respectively. The value of $H(p)$ when p is brought in to the cache is $L_1 + c(p)/s(p)$. p can only be evicted if L equals $H(p)$. Since $H(p)$ can only increase during the time that p is in the cache, we know that $L_2 - L_1 \geq c(p)/s(p)$.

Draw an interval on the real line from L_1 to L_2 that is closed on the left end and open on the right end. Assign the interval a weight of $s(p)$. If we draw an interval for every such eviction, the cost of GreedyDual-Size is upper bounded by the sum over all intervals of their length times their weight. By definition, all intervals lie in the range from 0 to L_{final}.

The final observation is that the total weight of all the intervals which contain any single point on the real line is at most s_{cache}. Consider a point L' on the line where an interval begins or ends. The total weight of the intervals that cover this point is the sum of the sizes of the documents which are in the cache when L reaches a value of L'. Since the size of the cache is at most s_{cache}, the sum of the weights of the intervals which cover L' is at most s_{cache}.

□

5 Performance Comparison

Using trace driven simulation, we compare the performance of GreedyDual-Size with LRU, Size, Hybrid, and LRV. Size, Hybrid, and LRV are all "champion" algorithms from previously published studies [WASAF96, LRV97, WA97]. In addition, for LRV, we first go through the whole trace to obtain the necessary parameters, thus giving it the advantage of perfect statistical information. In contrast, GreedyDual-Size takes into account cost, size and locality in a more natural manner and does not require tuning to a particular set of traces.

5.1 Performance Metrics

We consider five aspects of web caching benefits: hit ratio, byte hit ratio, latency reduction, hop reduction, and weighted-hop reduction. By hit ratio, we mean the number of requests that hit in the proxy cache as a percentage of total requests. By byte hit ratio, we mean the number of bytes that hit in the proxy cache as the percentage of the total number of bytes requested. By latency reduction, we mean the percentage of the sum of downloading latency for the pages that hit in cache over the sum of all downloading latencies. Hop reduction and weighted-hop reduction are used to measure the effectiveness of the algorithm at reducing network costs, as explained below.

To investigate the regulatory role that can be played by proxy caches, we model the network cost associated with each document as "hops". The "hops" value can be the number of network hops travelled by a document, to model the case when the proxy tries to reduce the overall load on Internet routers, or it can be the monetary cost associated with fetching the document, to model the case when the proxy has to pay for documents travelling through different network carriers.

We evaluate the algorithms in a situation where there is a skew in the distribution of "hops" values among the documents. The skewed distribution models the case when a particular part of the network is congested, or the proxy has to pay a different amount of money for documents travelling through different networks (for example, if the proxy is at an Internet Service Provider). In our particular simulation, we assign each Web server a hop value equal to 1 or 32 [2], so that 1/8 of the servers have hop value 32 and 7/8 of the servers have hop value 1. This simulates the scenario, for example, that 1/8 of the web servers contacted are located in Asia, or can only be reached through an expensive or congested link.

Associated with the "hop" value are two metrics: hop reduction and weighted-hop reduction. Hop reduction is the ratio between the total number of the hops of cache hits and the total number of the hops of all accesses; weighted-hop reduction is the corresponding ratio for the total number of hops times "packet savings" on cache hits. A cache hit's packet saving is $2 + file\_size/536$, as an estimate of the actual number of network packets required if the request is a cache miss (1 packet for the request, 1 packet for the reply, and $size/536$ for extra data packets, assuming a 536-byte TCP segment size).

For each trace, we first calculate the benefit obtained if the cache size is infinite. The values for all traces are shown in Table 1. In the table, BU-272 and BU-B19 are two sets of traces from Boston University [CBC95], VT-BL, VT-C, VT-G, VT-U are four sets of traces from Virginia Tech [WASAF96], DEC-U1:8/29-9/4 through DEC-U1:9/19-9/22 are the requests made by users 0-512 (user group 1) for each week in the three and half week period, and DEC-U2:8/29-9/4 through DEC-U2:9/19-9/22 are the traces for users 1024-2048 (user group 2). We experimented with other subsets of DEC traces and the results are quite similar to those obtained from these subsets.

[2]These numbers are chosen partly because, at one time, the maximum number of hops along a packet's route was 32.

Below, we divide our results into three subsections. In Section 5.2, we measure the hit rate and byte hit rate of different algorithms. In Section 5.3 we compare the latency reduction. In Section 5.4 we compare the hop reduction and weighted hop reduction. The corresponding value under the infinite cache are listed in Table 1. Since these simulations assume limited cache storage, their ratios cannot be higher than the infinite cache ratios.

The cache sizes investigated in the simulation were chosen by taking a fixed percentage of the total sizes of all distinct documents requested in the sequence. The percentages are 0.05%, 0.5%, 5%, 10% and 20%. For example, for trace DEC-U1:8/29-9/4, which includes the requests made by users 0-512 for the week of 8/29 to 9/4 and has a total data set size of 9.21GB, the cache sizes experimented are 4.6MB, 46.1MB, 461MB, 921MB and 1.84GB.

Due to space limitation, we organize the traces into four groups: Boston University traces, Virginia Tech traces, DEC-U1 traces, and DEC-U2 traces, and present the averaged results per trace group. The results for individual traces are similar.

5.2 Hit Rate and Byte Hit Rate

We introduce two versions of the GreedyDual-Size algorithm, GD-Size(1) and GD-Size(packets). GD-Size(1) sets the cost for each document to 1, and GD-Size(packets) sets the cost for each document to $2+size/536$ (that is, the estimated number of network packets sent and received if a miss to the document happens). In other words, GD-Size(1) tries to minimize miss ratio, and GD-Size(packets) tries to minimize the network traffic resulting from the misses.

Figure 6(a) shows the average hit ratio of the four groups of traces under LRU, Size, LRV, GD-Size(1), and GD-Size(packets). The graphs from left to right show the results for Boston University traces, Virginia Tech traces, DEC-U1 traces and DEC-U2 traces, respectively. Figure 6(b) is a simplified version of 6(a) showing only the curves of LRU and GD-Size(1), highlighting the differences between the two. Graph (c) shows the average byte hit ratio for the four trace groups under the different algorithms.

The results show that clearly, GD-Size(1) achieves the best hit ratio among all algorithms across traces and cache sizes. It approaches the maximal achievable hit ratio very fast, being able to achieve over 95% of the maximal hit ratio when the cache size is only 5% of the total data set size. It performs particularly well for small caches, suggesting that it would

Trace	Clients	Total Requests	Total GBytes	Hit Rate	Byte HR	Reduced Latency	Reduced Hops	Reduced WeightedHops
BU-272	5	17007	0.39	0.25	0.15	0.13	0.16	0.09
BU-B19	32	118104	1.59	0.47	0.27	0.20	0.48	0.25
VT-BL	59	53844	0.674	0.43	0.33	-	0.35	0.16
VT-C	26	11250	0.159	0.45	0.38	-	0.33	0.15
VT-G	26	47802	0.630	0.50	0.30	-	0.49	0.31
VT-U	74	164160	2.30	0.46	0.33	-	0.40	0.25
DEC-U1:8/29-9/4	512	633881	9.21	0.42	0.35	0.24	0.34	0.25
DEC-U1:9/5-9/11	512	691211	9.32	0.40	0.31	0.23	0.32	0.23
DEC-U1:9/12-9/18	512	658166	9.23	0.39	0.31	0.19	0.39	0.32
DEC-U1:9/19-9/22	512	280087	3.86	0.38	0.31	0.16	0.25	0.21
DEC-U2:8/29-9/4	1024	455858	5.57	0.33	0.22	0.20	0.27	0.19
DEC-U2:9/5-9/11	1024	428719	5.13	0.30	0.21	0.18	0.25	0.16
DEC-U2:9/12-9/18	1024	408503	4.94	0.29	0.19	0.15	0.24	0.17
DEC-U2:9/19-9/22	1024	170397	2.00	0.26	0.19	0.15	0.17	0.11

Table 1: Benefits under a cache of infinite size for each trace, measured as hit ratio, byte hit ratio, latency reduction, hop reduction, and weighted-hop reduction.

be a good replacement algorithm for main memory caching of web pages.

However, Figure 6(c) reveals that GD-Size(1) achieves its high hit ratio at the price of lower byte hit ratio. This is because GD-Size(1) considers the saving for each cache hit as 1, regardless of the size of document. GD-Size(packets), on the other hand, achieves the overall highest byte hit ratio and the second highest hit ratio (only moderately lower than GD-Size(1)). GD-Size(packets) seeks to minimize (estimated) network traffic, in which both hit ratio and byte hit ratio play a role.

For the Virginia Tech traces, LRV outperforms GD-Size(packets) in terms of hit ratio and byte hit ratio. This is due to the fact that those traces have significant skews in the probability of references to different sized files, and LRV knows the distribution before-hand and includes it in the calculation. However, for all other traces where the skew is less significant, LRV performs worse than GD-Size(packets) in terms of both hit ratio and byte hit ratio, despite its heavy parameterization and foreknowledge.

LRU performs better than SIZE in terms of hit ratio when the cache size is small (less or equal than 5% of the total date set size), but performs slightly worse when the cache size is large. The relative comparison of LRU and Size differs from the results in [WASAF96], but agrees with those in [LRV97].

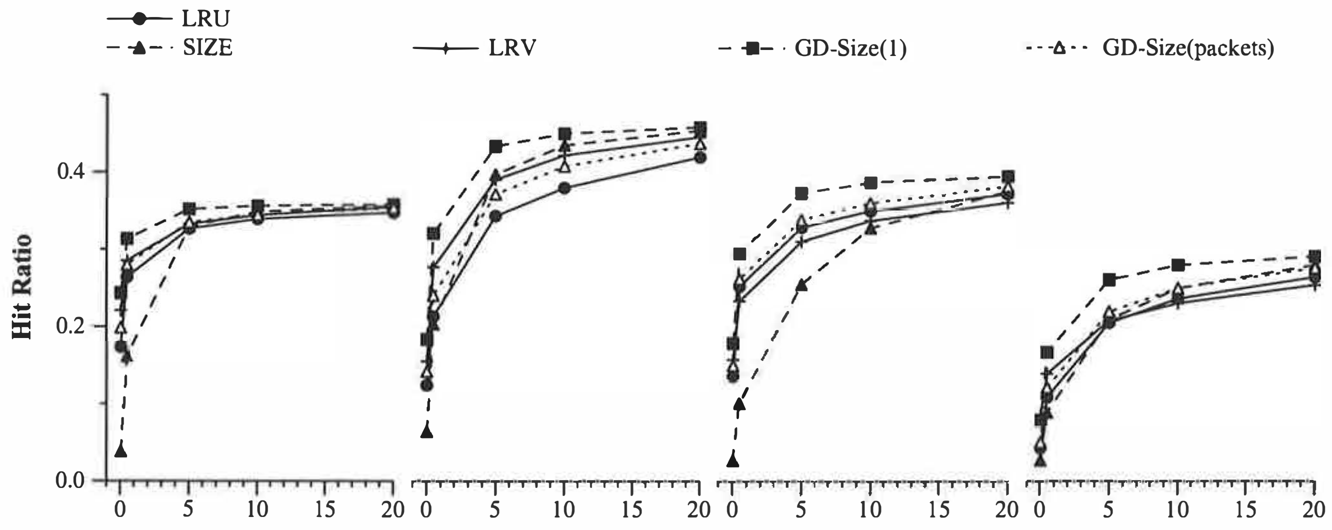
In summary, for proxy designers that seek to maximize hit ratio, GD-Size(1) is the appropriate algorithm. If both high hit ratio and high byte hit ratio are desired, GD-Size(packets) is the appropriate algorithm.

5.3 Reduced Latency

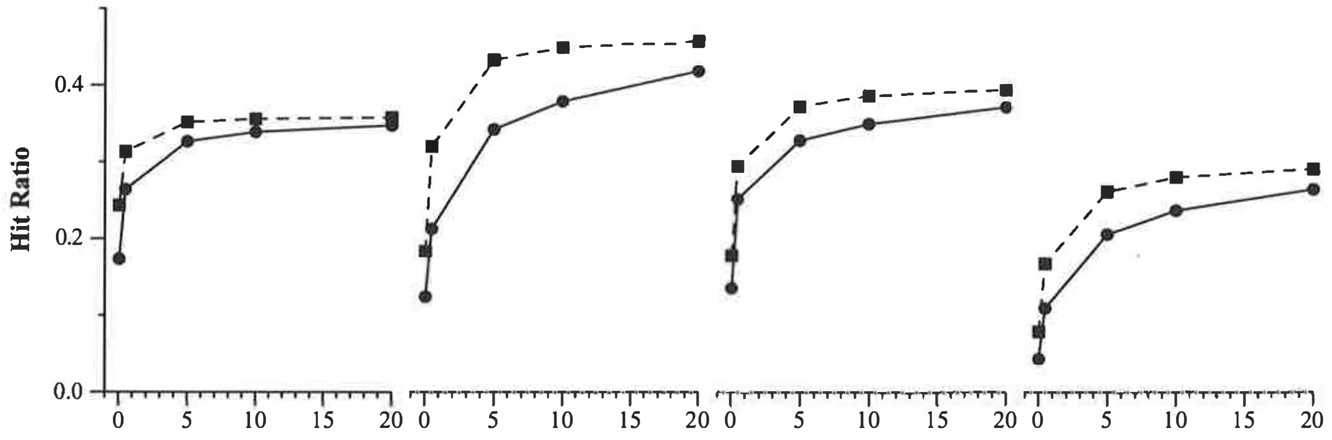
Another major concern for proxies is to reduce the latency of HTTP requests through caching, as numerous studies have shown that the waiting time has become the primary concern of Web users. One study [WA97] introduced a proxy replacement algorithm called *Hybrid*, which takes into account the different latencies incurred to load different web pages, and attempts to minimize the average latency. The study [WA97] further showed that in general the algorithm has a lower average latency than LRU, LFU and SIZE.

We also designed two versions of GreedyDual-Size that take latency into account. One, called GD-Size(latency), sets the cost of a document to the latency that was required to download the document. The other, called GD-Size(avg_latency), sets the cost to the estimated download latency of a document, using the same method of estimating latency as in Hybrid [WA97].

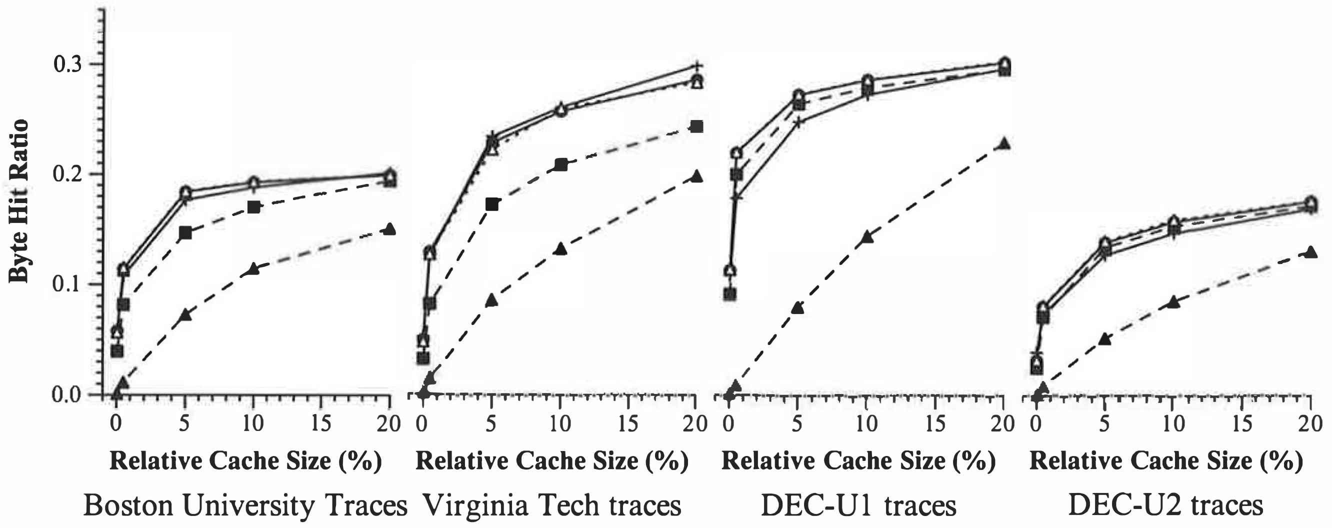
Figure 7(a) shows the latency reductions for LRU, Hybrid, GD-Size(1), GD-Size(latency) and GD-Size(avg_latency). The graphs, from left to right, show the results for Boston University traces, DEC-U1 traces and DEC-U2 traces. The figure unfortunately does not include results for Virginia Tech traces because they do not have latency information for each HTTP request. Clearly, GD-Size(1) performs the best, yielding the highest latency re-



(a) Hit ratios of LRU, Size, LRV, GD-Size(1) and GD-Size(packets) for each trace group.



(b) A simplified version of (a) showing only the curves for LRU and GD-Size(1).



(c) Byte hit ratios of LRU, Size, LRV, GD-Size(1), and GD-Size(packets) for each trace group.

Figure 6: Hit ratio and byte hit ratio comparison of the algorithms.

duction. GD-Size(latency) and GD-Size(packets) finish the second, with LRU following close behind. GD-Size(avg_latency) performs badly for small cache sizes, but performs very well for relatively large cache sizes. Finally, Hybrid performs the worst.

Examination of the results shows that the reason for Hybrid's poor performance is its low hit ratio. In the Boston University traces, Hybrid's hit ratio is much lower than LRU's for cache sizes $\leq$ 5% of the total data set sizes, and only slightly higher for larger cache sizes. For all DEC traces, Hybrid's hit ratio is much lower than LRU's, under all cache sizes. Hybrid has a low hit ratio because it does not consider how recently a document has been accessed during replacement.

Since [WA97] reports that Hybrid performs well, our results here seem to suggest that Hybrid's performance is perhaps trace-dependent. In our simulation of Hybrid we used the same constants in [WA97], without tuning them to our traces. Unfortunately we were not able to obtain the traces used in [WA97].

It is a surprise to us that GD-Size(1), which does not take latency into account, performs better than GD-Size(latency) and GD-Size(avg_latency). Detailed examination of the traces shows that the latency of loading the same document varies significantly. In fact, for each of the DEC traces, variance among latencies of the same document ranges from 5% to over 500%, with an average around 71%. Thus, a document that was considered cheap (taking less time to download) may turn out expensive at the next miss, while a document that was considered expensive may actually take less time to download. The best bet for the replacement algorithm, it seems, is to maximize hit ratio.

In summary, GD-Size(1) is the best algorithm to reduce average latency. The high variance among loading latencies for the same document reduces the effectiveness of latency-conscious algorithms.

5.4 Network Costs

To incorporate network cost considerations, GD-Size(hops) sets the cost of each document to the hop value associated with the Web server of the document, and GD-Size(weightedhops) sets the cost to be $hops \times (2 + file\_size/536)$. Figure 7(b) and 7(c) show the hop reduction and weighted-hop reduction for LRU, GD-Size(1), GD-Size(hops), and GD-Size(weightedhops).

The results show that algorithms that consider network costs do perform better than algorithms that are oblivious to them. The results here are different from the latency results because the network cost associated with a document does not change during our simulation. The results also show that the specifically designed algorithms achieve their effect. For hop reduction, GD-Size(hops) performs the best, and for weighted-hop reduction, GD-Size(weightedhops) performs the best. This shows that GreedyDual-Size not only can combine cost concerns nicely with size and locality, but is also very flexible and can accommodate a variety of performance goals.

Thus, we recommend GD-Size(hops) as the replacement algorithm for the regulatory role of proxy caches. If the network cost is proportional to the number of bytes or packets, then GD-Size(weightedhops) is the appropriate algorithm.

5.5 Summary

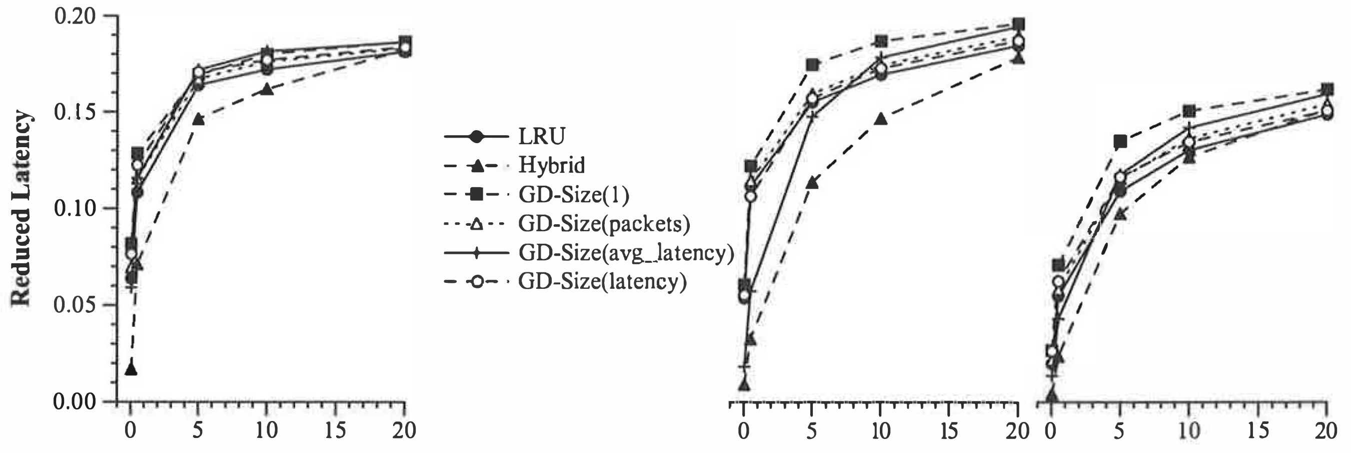
Based on the above results, we have the following recommendation. If the proxy wants high hit ratio or low average latency, GD-Size(1) is the appropriate algorithm. If the proxy desires high byte hit ratio as well, then GD-Size(packets) achieves a good balance among the different goals. If the documents have associated network or monetary costs that do not change over time, or change slowly over time, then GD-Size(hops) or GD-Size(weightedhops) is the appropriate algorithm. Finally, in the case of main memory caching of web documents, GD-Size(1) should be used because of its superior performance under small cache sizes.

6 Conclusion

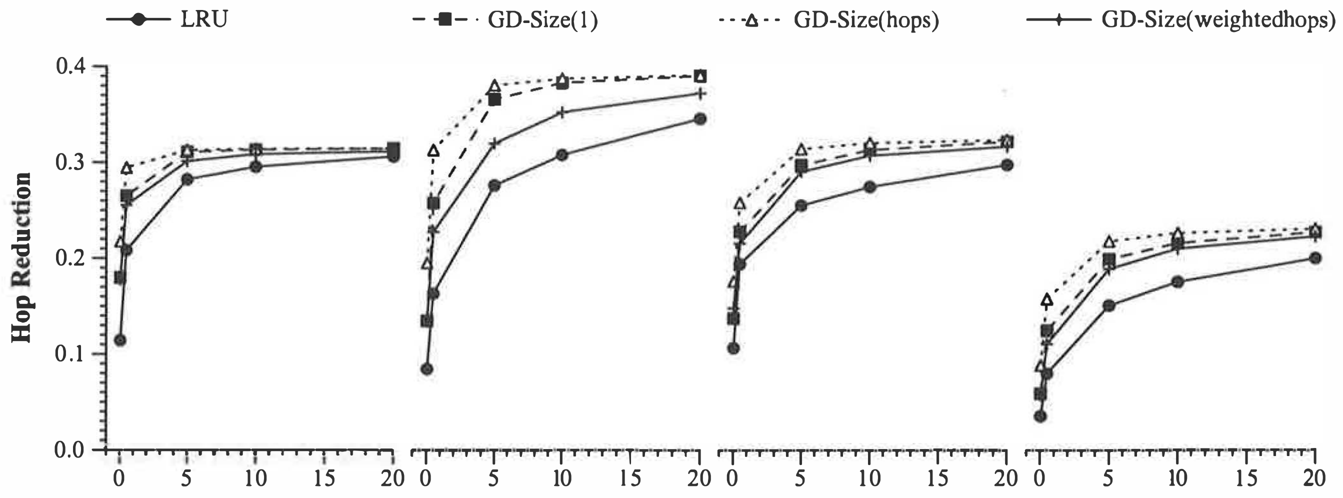
This paper introduces a simple web cache replacement algorithm: GreedyDual-Size, and shows that it outperforms existing replacement algorithms in many performance aspects, including hit ratios, latency reduction, and network cost reduction. GreedyDual-Size combines locality, cost and size considerations in a unified way without using any weighting function or parameter. It is simple to implement and accommodates a variety of performance goals. Through trace-driven simulations, we identify the cost definitions for GreedyDual-Size that maximize different performance gains. GreedyDual-Size can also be applied to main memory caching of Web documents to further improve performance.

The GreedyDual-Size algorithms shown so far can only optimize one performance measure at a time. We are looking into how to adjust the algorithm when the goal is to optimize more than one performance measures (for example, both hit ratio and byte hit ratio). We also plan to study the integration of hint-based prefetching with the cache replacement algorithm.

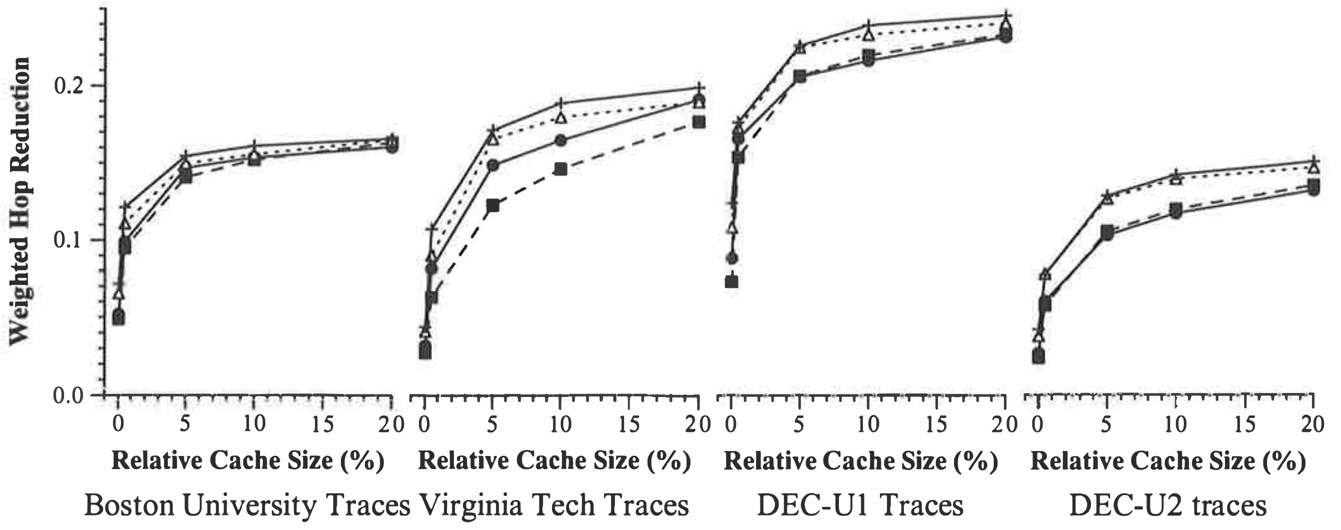
Finally, we have shown that if an appropriate



(a) Latency reduction under LRU, Hybrid, GD-Size(1), GD-Size(packets), GD-Size(latency) and GD-Size(avg_latency) for Boston University, DEC-U1, and DEC-U2 traces.



(b) Hop reduction under LRU, GD-Size(1), GD-Size(hops) and GD-Size(weightedhops).



(c) Weighted hop reduction under LRU, GD-Size(1), GD-Size(hops) and GD-Size(weightedhops).

Figure 7: Latency, hops, and weighted hops reductions under various algorithms.

network cost can be associated with a document, GreedyDual-Size algorithm can be used to adjust the caching of different documents to affect the Web traffic. In other words, if proxy caches use the GreedyDual-Size algorithm, and they can be informed of the congestion on the network, the caches can cooperate to reduce the traffic over the congested links. However, how to detect congested path on the network and how to assign appropriate cost values for the affected documents are topics beyond the scope of this paper, and remain our future work.

Acknowledgement

The research is not possible without the support from people who make their proxy traces available. Sandy Irani is supported in part by NSF Grant CCR-9309456. Our shepherd, Carl Staelin, contributed the accumulative percentage graphs and the quartile graphs and greatly improved the paper. Finally, the anonymous referees provided very helpful comments.

References

[ASAWF95] M. Abrams, C.R. Standbridge, G.Abdulla, S. Williams and E.A. Fox. Caching Proxies: Limitations and Potentials. WWW-4, Boston Conference, December, 1995.

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[CD73] G. Coffman, Jr., Edward and Peter J. Denning, Operating Systems Theory, *Prentice-Hall, Inc.* 1973.

[CKPV91] M. Chrobak, H. Karloff, T. H. Payne and S. Vishwanathan. New results on server problems. newblock *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991.

[DEC96] Digital Equipment Cooperation, Digital's Web Proxy Traces ftp://ftp.digital.com/pub/DEC/traces/proxy /webtraces.html.

[FKIP96] A. Feldman, A. Karlin, S. Irani, S. Phillips. Private Communication.

[Ho97] Hosseini, Saied, Private Communication.

[LC97] Chengjie Liu, Pei Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the 1997 International Conferences on Distributed Computing Systems*, May, 1997.

[Ir97] S. Irani. Page replacement with multi-size pages and applications to web caching. In the *Proceedings for the 29th Symposium on the Theory of Computing*, 1997, pages 701-710.

[LRV97] P. Lorenzetti, L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/ luigi/research.html.

[CBC95] Carlos R. Cunba, Azer Bestavros, Mark E. Crovella Characteristics of WWW Client-based Traces BU-CS-96-010, Boston University.

[LM96] Paul Leach and Jeff Mogul. The Hit Metering Protocol. Manuscript.

[HT97] IETF The HTTP 1.1 Protocol - Draft. http://www.ietf.org.

[ST85] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[Tufte] Edward Tufte The Visual Display of Quantitative Information. Graphics Printers, Feburary 1992.

[W3C] The Notification Protocol. http://www.w3c.org.

[WASAF96] S. Williams, M. Abrams, C.R. Standbridge, G.Abdulla and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM Sigcomm96*, August, 1996, Stanford University.

[WA97] R. Wooster and M. Abrams. Proxy Caching the Estimates Page Load Delays. In the *6th International World Wide Web Conference,* April 7-11, 1997, Santa Clara, CA. http://www6.nttlabs.com/HyperNews/get/ PAPER250.html.

[WPB] Jussara Almeida and Pei Cao. The Wisconsin Proxy Benchmark (WPB). http://www.cs.wisc.edu/~cao/wpb1.0.html.

[You91b] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*,June 1994, vol. 11,(no.6):525-41. Rewritten version of "On-line caching as cache size varies", in The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250, 1991.

[You97] N. Young. Online file caching. To appear in *the Proceedings for the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace

Steven D. Gribble and Eric A. Brewer
The University of California at Berkeley
{gribble,brewer}@cs.berkeley.edu

Abstract

In this paper, we present the analysis of a large client-side web trace gathered from the Home IP service at the University of California at Berkeley. Specifically, we demonstrate the heterogeneity of web clients, the existence of a strong and very predictable diurnal cycle in the clients' web activity, the burstiness of clients' requests at small time scales (but not large time scales, implying a lack of self-similarity), the presence of locality of reference in the clients' requests that is a strong function of the client population size, and the high latency that services encounter when delivering data to clients, implying that services will need to maintain a very large number of simultaneously active requests. We then present system design issues for Internet middleware services that were drawn both from our trace analysis and our implementation experience of the TranSend transformation proxy.

1 Introduction

The growth of the Internet, and particularly of web-oriented middleware services ([15], [3], [6]) within the Internet, has seen a recent explosion [31]. These middleware services, particularly the more popular services that experience extremely high load, must overcome a number of challenging system design issues in order to maintain fast response time, constant availability, and capacity. Services must be able to accommodate an increasingly varied client population (in terms of hardware, software, and network connectivity). They must be able to handle offered loads of hundreds of requests per second, and because of the often slow connectivity to clients and the implied lengthy delivery times, they must be able to handle hundreds of simultaneously outstanding tasks.

Previous work has explored the performance of operating system primitives and the relationship between OS performance and architecture ([29], [2]), and operating system design issues for busy Internet services ([19], [27]). In contrast, this paper raises a number of system design issues specifically for Internet middleware services. These issues were encountered during two separate but related efforts: the analysis of a set of extensive client-side HTTP [5] traces that we gathered from the University of California at Berkeley's dial-in modem banks during October and November of 1996, and the implementation and deployment experience we gained from the TranSend Internet middleware service [15].

Since nearly 70% of all Internet clients use dial-in modems of speeds of 28.8 Kb/s or less [18], we use the traces to make a number of observations about the Internet user population and the services with which they communicate. Section 2 discusses the gathering of the traces, including the tools used and the information gathered, and section 3 performs a detailed analysis of these traces, both in terms of observations made about the client population and the services themselves. In section 4, we discuss the middleware system design issues drawn from our experience with the TranSend transformation proxy service, and in section 5 we present related work. Finally, in section 6 we conclude.

2 Home IP Trace Gathering

During October and November of 1996, we gathered over 45 consecutive days worth of HTTP traces from the Home IP service offered by UC Berkeley to its students, faculty, and staff available to researchers. (Two and a half weeks worth of anonymized versions of these traces have been made available at http://www.acm.org/ita.) Home IP provides dial-up PPP/SLIP connectivity using 2.4 kb/s, 9.6 kb/s, 14.4 kb/s, or 28.8 kb/s wireline modems, or Metricom Ricochet wireless modems (which achieve approximately 20-30 kb/s throughput with a 500 ms RTT).

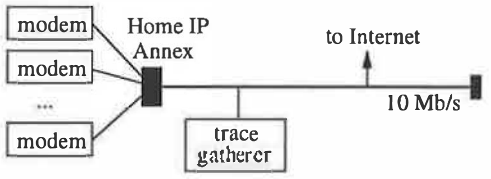


Figure 1: **The Home IP Tracing Environment**

2.1 IPSE

The HTTP client traces were unobtrusively gathered through the use of a packet sniffing machine placed on a 10 Mb/s Ethernet segment at the headend of the Home IP modem bank through which all IP traffic flowed (figure 1). The trace gathering program that we used was a custom HTTP module written on top of the Internet Protocol Scanning Engine (IPSE)[17]. IPSE is a user-level packet filter that runs on Linux; IPSE allows filter modules to capture TCP segments and recreate the TCP streams observed by the endpoints of the TCP connection. The custom module was therefore able to parse each HTTP request as it was happening, and write out the salient features of each HTTP request to a log file on-the-fly. Only traffic destined for port 80 was traced; all non-HTTP protocols and HTTP connections to other ports were excluded. Each user of the Home IP service is assigned a static IP address, so we could track individual users over the entire duration of the tracing experiment.

2.2 The Trace Files

The 45 day trace contains approximately 24,000,000 HTTP requests, representing the web surfing behaviour of over 8,000 unique clients. The trace capture tool collected the following information for each HTTP request seen:

- the time at which the client made the request, the time that the first byte of the server response was seen, and the time that the last byte of the server response was seen,
- the client and server IP addresses and ports,
- the values of the **no-cache**, **keep-alive**, **cache-control**, **if-modified-since**, **useragent**, and **unless** client headers (if present),
- the values of the **no-cache**, **cache-control**, **expires**, and **last-modified** server headers (if present),
- the length of the response HTTP header and response data, and
- the request URL.

IPSE wrote this information to disk in a compact, binary form. Every four hours, IPSE was shut down and restarted, as its memory image would get extremely large over time due to a memory leak that we were unable to eliminate. This implies that there are two potential weaknesses in these traces:

1. Any connection active when the engine was brought down will have a possibly incorrect timestamp for the last byte seen from the server, and a possibly incorrect reported size.
2. Any connection that was forged in the very small time window (about 300 milliseconds) between when the engine was shut down and restarted will not appear in the logs.

We estimate that no more than 150 such entries (out of roughly 90,000-100,000) are misreported for each 4 hour period.

3 Trace Analysis

In this section, we present the results of our analysis of the Home IP traces. In section 3.1, we demonstrate the heterogeneity of the observed client population. Section 3.2 discusses the request rates and interarrival times generated by the client population. In 3.3, object type and size distributions are presented. Section 3.4 demonstrates the existence of locality of reference within the traces through the use of a number of cache simulations driven by trace entries. Finally, section 3.5 presents distributions of service response times, and argues that at any given time, a very large number of outstanding requests will be flowing through middleware or end services.

3.1 Client Heterogeneity

Table 1 lists the most frequently observed "User-Agent" HTTP headers observed within the Home IP traces. From this table, it is easy to make a common misconclusion about web clients, namely that the set of web clients in use is extremely homogeneous, as nearly all browsers observed in our traces are either the Netscape Navigator [28] or Microsoft Internet Explorer (MSIE) [26] browsers running on the Windows or Macintosh operating systems. However, there is significant heterogeneity arising from the many versions of these browsers and their widely varying feature sets. Furthermore, we observed a total 166 different UserAgent values within the traces, representing a wide range of desktop systems (MacOS, Win16, NetBSD, Linux, etc.) More significantly, however, we saw requests from a number of exotic clients such as Newton PDAs running the NetHopper [1] browser.

Browser	OS	% Seen
Netscape	Windows 95	55.1
	Macintosh	19.7
	Windows (other)	8.8
	Windows NT	3.5
	Linux	2.2
	Other	0.4
MSIE	Windows 95	7.6
	Macintosh	0.6
	Windows NT	0.7
	Windows (other)	0.1
Other		1.3

Table 1: **UserAgent HTTP headers:** this table lists the 10 most frequent UserAgent headers observed in the traces. "Other" browsers observed include PointCast, Cyberdog, Mosaic, Opera, Lynx, JDK, and NetHopper.

Internet services that do not want to limit the effective audience of their content must therefore be able to deliver content that suits the needs of all of these diverse clients. Either the services themselves must adapt their content, or they must rely on the emergence of middleware services (such as in [13], [14], and [7]) to adapt content on the fly to better suit the clients' particular needs.

3.2 Client Activity

As seen in figure 2, the amount of activity seen from the client population is strongly dependent on the time of day. The Berkeley web users were most active between 8:00pm and 2:00am, with nearly no activity seen at 7:00am. Services that receive requests from local users can thus expect to have widely varying load throughout the day; internationally used services will most probably see less of a strong diurnal cycle. Other details can be extracted from these graphs. For example, there is a decrease of activity at noon and at 7:00pm, presumably due to lunch breaks and dinner breaks, respectively.

The diurnal cycle is largely independent of the day of the week, but there are some minor differences: for instance, on Fridays and Saturdays, the traffic peaks are slightly higher than during the rest of the week. However, the gross details of the traces remain independent of the day of the week. We calculated the average daily cycle observed by averaging the number of events seen per minute for each minute of the day across 15 days of traffic. For our calculation, we picked days during which there were no anomalous trace effects, such as network outages. Figure 3 shows this average cycle, including a polynomial curve fit that can be used to calculate approximate load throughout a typical day.

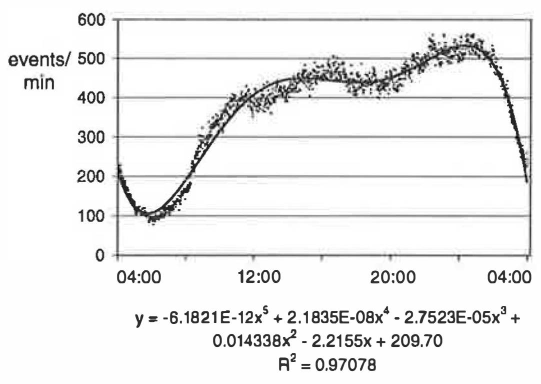


Figure 3: **Average diurnal cycle** observed within the traces - each minutes worth of activity shown is the average across 15 days worth of trace events. The y-axis shows the average number of observed requests per minute.

On shorter time scales, we observed that client activity was less regular. Figure 4 illustrates the observed request rate at three time scales from a one-day segment of the traces. At the daily and hourly time scales, traffic is relatively smooth and predictable - no large bursts of activity are present. At the scale of tens of seconds, very pronounced bursts of activity can be seen; peak to average ratios of more than 5:1 are common.

Many studies have explored the self-similarity of network traffic ([4], [16], [21], [22], [24], [30]), including web traffic [9]. Self-similarity implies burstiness at all timescales - this property is **not** compatible with our observations. One indicator of self-similarity is a heavy-tailed interarrival process. As figure 5 clearly shows, the interarrival time of GIF requests seen within the traces is exponentially distributed, and therefore not heavy tailed. (We saw similar exponential distributions for other data types' request processes, as well as for the aggregate request traffic.) These observations correspond to

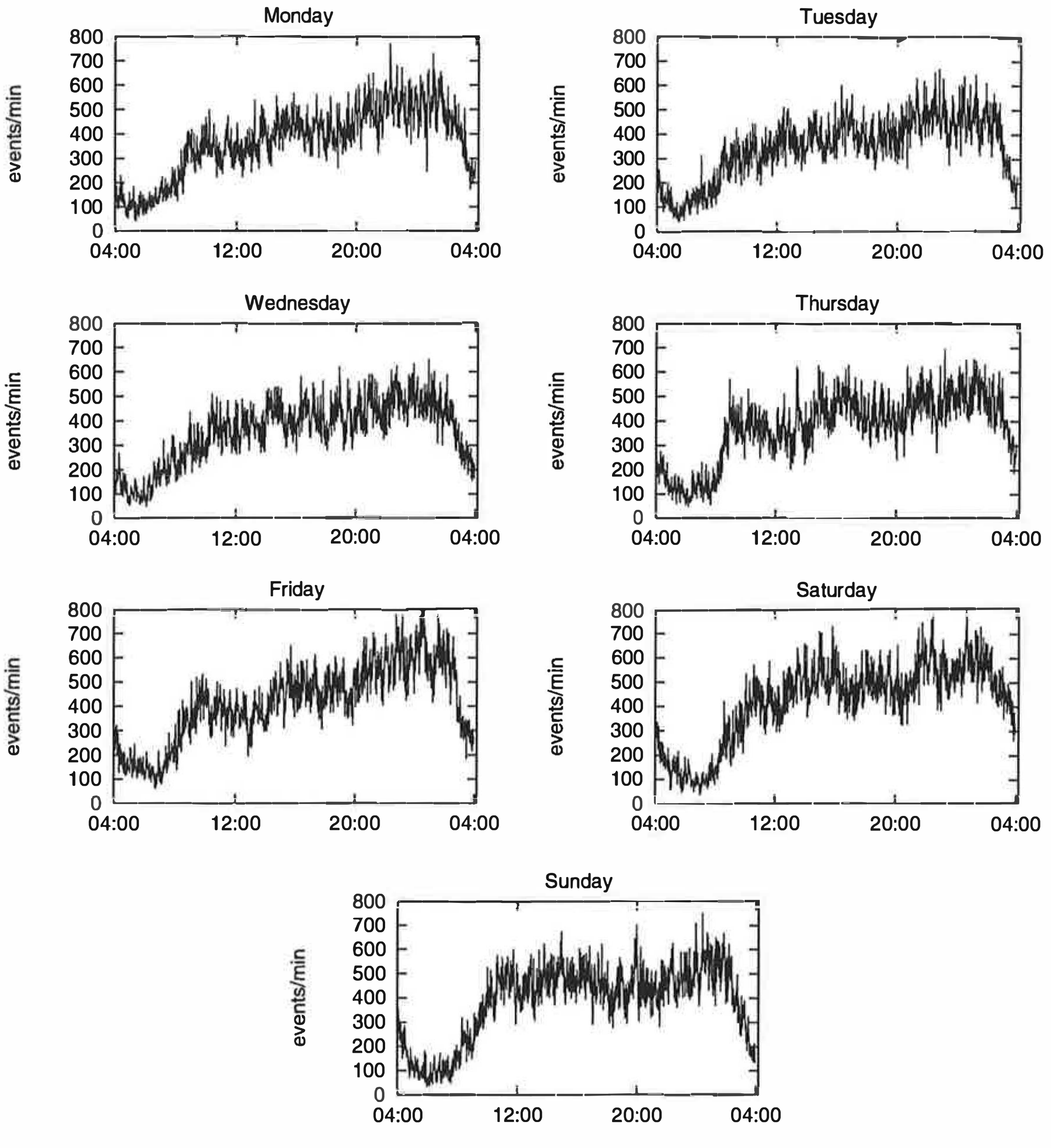


Figure 2: **Diurnal cycle** observed within the traces - each graph shows 1 day worth of trace events. The y-axis shows the number of observed requests per minute.

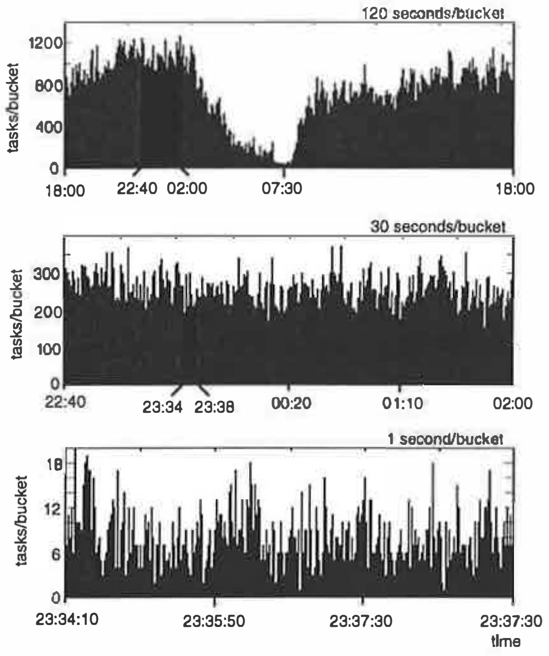


Figure 4: **Request rate** observed over a 24 hour, 3 hour, and 3 minute period of the traces.

requests generated from a large population of independent users.

Internet services must be able to handle rapidly varying and bursty load on fine time scales (on the order of seconds), but these bursts tend to smooth themselves out on larger time scales (on the order of minutes, hours, or days). The provisioning of resources for services is therefore somewhat simplified.

3.3 Reference type and size distributions

Section 3.1 answered the question of who is requesting data, and section 3.2 discussed how often data is requested. In this section, we inspect the nature of the data that is requested. Figure 6a shows the mime type breakdown of the transferred data in terms the number of bytes transferred, 6b shows this breakdown in term of files transferred.

From figure 6a, we see that most of the bytes transferred over the Home IP modem lines come from three predominant mime types: **text/html**, **image/gif**, and **image/jpeg**. Similarly, figure 6b shows that most files sent over the modem lines have the same three predominant mime types. Interestingly, however, we see that although most bytes transferred correspond to JPEG images, most files transferred correspond to GIF images. This means

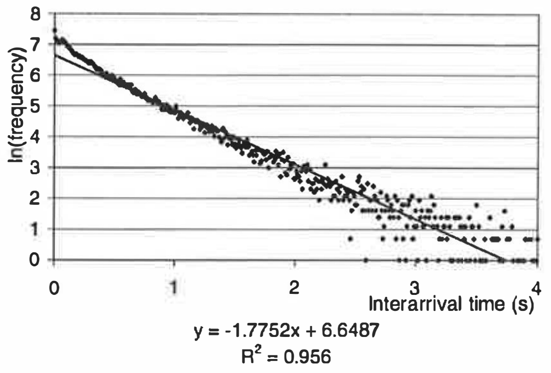


Figure 5: **Interarrival time distribution** for GIF data type requests seen within a day-long trace portion. Note that the Y-axis is on a logarithmic scale.

that, on average, JPEGs are larger than GIFs.

The fact that nearly 58% of bytes transferred and 67% of files transferred are images is good news for Internet cache infrastructure proponents. Image content tends to change less often than HTML content - images are usually statically created and have long periods of stability in between modification, in comparison to HTML which is becoming more frequently dynamically generated.

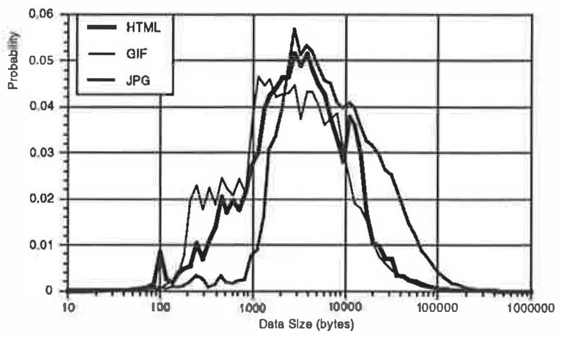


Figure 7: **Size distributions by MIME type**, shown on a logarithmic scale. The average HTML file size is 5.6 kilobytes, the average GIF file size is 4.1 kilobytes, and the average JPEG file size is 12.8 kilobytes.

In figure 7, we see the distribution of sizes of files belonging to the three most common mime type. Two observations can immediately be made: most Internet content is less than 10 kilobytes in size, and data type size distributions are quite heavy-tailed, meaning that there is a non-trivial number of large

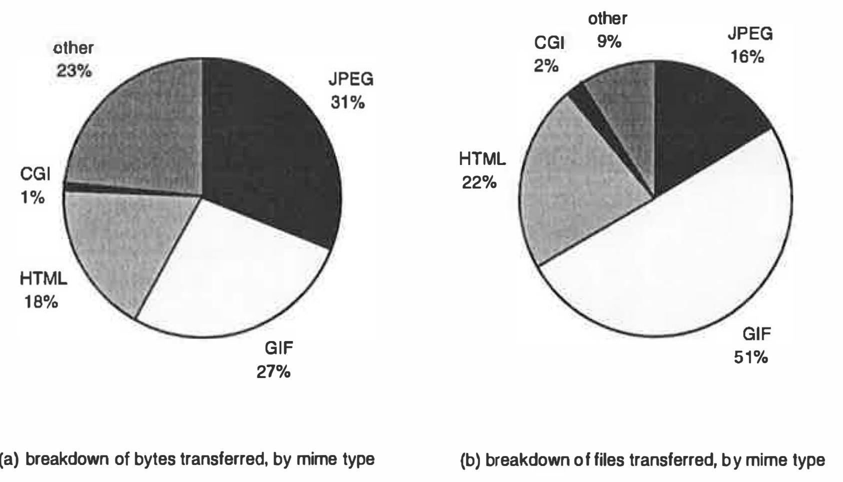


Figure 6: **Breakdown of bytes and files transferred by MIME type**

data files on the web. Looking more closely at individual distributions, we can confirm our previous hypothesis that JPEG files tend to be larger than GIF files. Also, the JPEG file size distribution is considerably more heavy-tailed than the GIF distribution. There are more large JPEGs than GIFs, perhaps in part because JPEGs tend to be photographic images, and GIFs tend to be cartoons, line art, or other such simple, small images.

There are other anomalies in these distributions. The GIF distribution has two visible plateaus, one at roughly 300-1000 bytes, and another at 1000-5000 bytes. We hypothesize that the 300-1000 byte plateau is caused by small "bullet" images or icons on web pages, and the 1000-5000 byte plateau represents all other GIF content, such as cartoons, pictures, diagrams, advertisements, etc. Another anomaly is the large spike in the HTML distribution at roughly 11 kilobytes. Investigation revealed that this spike is caused by the extremely popular Netscape Corporation "Net Search" page.

3.4 Locality of Reference

A near-universal assumption in systems is that of locality of reference, and the typical mechanism used to take advantage of this locality of reference is caching ([11], [8]). The effectiveness of caching depends upon a number of factors, including the size of the user population that a cache is serving and the size of the cache serving that population.

To measure the effectiveness of infrastructure caching (as opposed to client-side or server-side caching) with respect to the HTTP references captured from the Home IP population, we implemented a cache simulator and played segments of the traces at these caches. We filtered out requests from all but a parameterizable set of client IP addresses in order to simulate client populations of different sizes. The cache simulator obeyed all HTTP cache pragmas (such as the no-cache pragma, the if-modified-since header, and the expiry header), and implemented a simple LRU eviction policy. Figure 8 shows measured cache hit rate as a function of cache size for different user population sizes, and figure 9 shows measured hit rate as a function of user population size for different cache sizes.

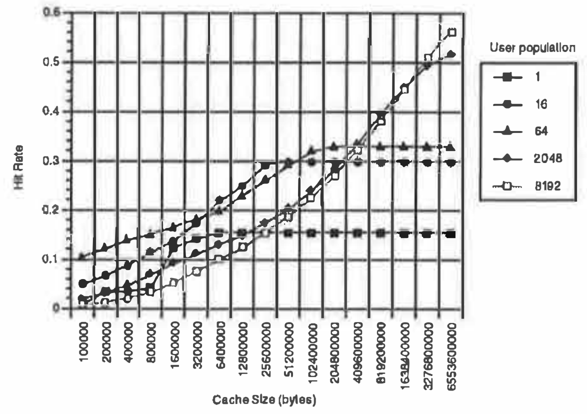


Figure 8: **Hit rate vs. Cache size** for a number of different user population sizes.

Figure 8 shows two trends: the first is that an increasingly large cache size results in an increasingly large cache hit rate. The second trend is that we observed that hit rate is a very strong function of the

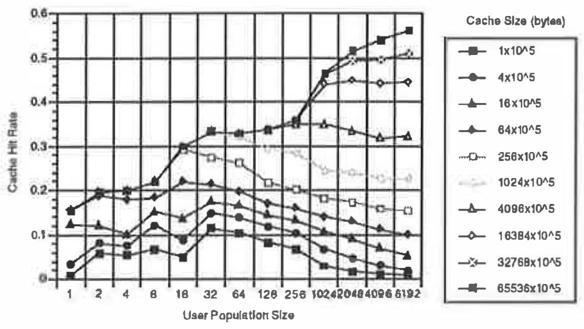


Figure 9: **Hit rate vs. User Population size** for a number of cache sizes.

user population size. As the population gets larger, the locality of reference within that population gets stronger, and caches become more effective. For a given population size, the cache hit rate as a function cache size plateaus at the working set size of that population. In figure 9, one additional trend can be observed: as the user population size grows, if the cache size does not also pace the increasingly large working set of that population, the cache hit rate will start to drop as the cache effectively begins to thrash from constant evictions.

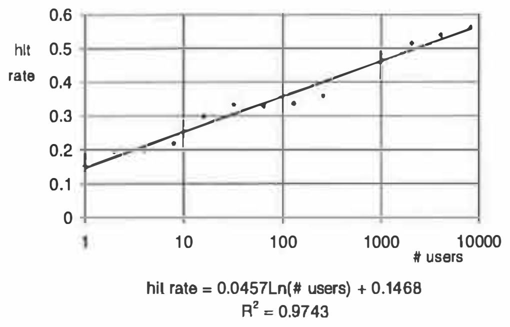


Figure 10: **Asymptotic Hit Rate vs. User Population Size**

An interesting question is: what is the maximum possible cache performance for a given user population size? In figure 10, we have plotted the asymptotic hit rate achieved in the limit of infinitely large cache size as a function of the user population size. In other words, this graph explicitly shows the cachable working set size of a given user population size. We see that for the range of population sizes that we can model from our traces, the asymptotic hit rate grows logarithmically with population size. Obviously, this logarithmic increase cannot continue forever, as there is a maximum possible hit rate of 100%; unfortunately, our traces do not contain a large enough population size to see the logarithmic increase tail off.

A factor that can alter the performance of Internet caches is the increasingly prevalent use of cache pragmas in HTTP headers. To investigate this effect, we measured the percentage of HTTP client requests and server responses that contained relevant headers, namely:

no-cache: This header can be supplied by either the client or the server, and indicates that the requested or returned data may not be served out of or stored in a client-side or proxy cache.

cache-control This is a generic, extensible HTTP header whose value contains the real directive. Cache-control is intended to be used to supply additional caching directives that are interpreted by middleware caches, rather than by the end server or client.

if-modified-since This HTTP header allows a client to specify that a document should be returned only if it has been modified after a certain date. If it hasn't, then the client uses a locally cached version.

expires This HTTP header allows a server to supply an expiry date for returned content. Caches obey this directive by treating cached data as stale if the expiration date has occurred.

last-modified This HTTP header allows a server to indicate when a document has last been modified. This is typically used as a hint for caches when calculating time-to-live (TTL) values, or when returning HTTP headers in response to a client's HEAD request.

As can be seen in table 2, most HTTP headers that can affect cache performance are rarely used. The most frequently used header is the last-modified server response header; this header is now commonly returned by default from most HTTP servers. The presence of this header in data stored within a middleware cache or end server can be compared to the value of the if-modified-since client header to test whether or not cached data is stale. Unfortunately, only 1/4 of the client requests contained this header. Cache-control, no-cache, and expiry headers are extremely infrequent. These headers should become more commonly used once HTTP 1.1 compliant browsers and servers are deployed.

Pragma occurrence	11/8/96	4/28/97
(C) no-cache	7.2%	5.7%
(C) cache-control	0%	0.004%
(C) if-modified-since	22.8%	20.6%
(S) no-cache	0.2%	0.5%
(S) cache-control	0.1%	0.5%
(S) expires	4.7%	5.0%
(S) last-modified	54.3%	54.5%

Table 2: **HTTP header frequencies:** this table summarizes the percent of HTTP client requests (C) and server responses (S) that contained various HTTP headers that affect caching behaviour.

Internet services can benefit quite strongly from caching, as there is significant locality in a user population's references. Services must be careful to deploy an adequately large cache in order to capture the working set of that population.

3.5 Service Response Times

The recently emerging class of middleware services must take into consideration the performance of conventional content-providing Internet services as well as the characteristics of the client population. Middleware services retrieve and transform content on behalf of clients, and as such interact directly with content-providing services, relying in part on the services' performance to determine their own.

In figure 11, we present a breakdown of the time elapsed during the servicing of clients' requests. Figure 11a shows the distribution of the elapsed time between the first byte of the client request and the first byte of the server's response observed by the trace gatherer, shown using both a linear and a logarithmic y-axis. This initial server reaction time distribution is approximately exponentially decreasing, with the bulk of reaction times being far less than a second. Internet services are thus for the most part quite reactive, but there is a significant number of very high latency services.

Figure 11b shows the distribution of the elapsed time between the first observed server response byte and the last observed server response byte (as measured by when the TCP connection to the server is shut down).[1] From these graphs, we see that complete server responses are usually delivered to the clients in less than ten seconds, although a great number of responses take many tens of seconds to deliver. (Bear in mind that the response data is being delivered over a slow modem link, so this is not too surprising.)

A number of anomalies can be seen in this graph, for instance the pronounced spikes at 0, 4, 30, and roughly 45 seconds. The spike at 0 seconds corresponds to HTTP requests that failed or returned no data. The spike at 4 seconds remains a bit of a mystery - however, note that the 4 second delivery time corresponds to 14 KB worth of data sent over a 28.8 KB modem, which is almost exactly the size of the "home_igloo.jpg" picture served from Netscape's home page, one of the most frequently served pages on the Internet. We believe that the spikes at 30 and 45 seconds most likely correspond to clients or servers timing out requests. Finally, figure 11b shows the distribution of total elapsed time until a client request is fully satisfied. This distribution is dominated by the time to deliver data over the clients' slow modem connections.

From these measurements, we can deduce that Internet servers and middleware services must be able to handle very large amounts of simultaneous, outstanding client requests. If a busy service expects to handle many hundreds of requests per second and requests take tens of seconds to satisfy, there will be many thousands of outstanding requests at any given time. Services must be careful to minimize the amount of state dedicated to each individual request the overhead incurred when switching between the live requests.

3.6 Summary

This section of the paper presented a detailed analysis of the Berkeley Home IP traces. We demonstrated the heterogeneity of the user population, the burstiness of traffic a fine-grained time scales, the presence of a strong and predictable diurnal traffic cycle, locality in client web requests, and the heavy-tailed nature of web service response times. In the next section, we discuss how these observations relate to a real Internet middleware service designed at Berkeley, the TranSend distillation proxy.

4 System Design Experience from TranSend

The TranSend middleware service provides distillation ([13], [14]) services for the Berkeley Home IP modem user population, representing roughly 8,000

[1]Persistent HTTP connections were very uncommon in these traces, but these special cases were handled correctly - the elapsed time until the last byte from the server for a given request is seen is reported in these figures.

(a)

(b)

(c)

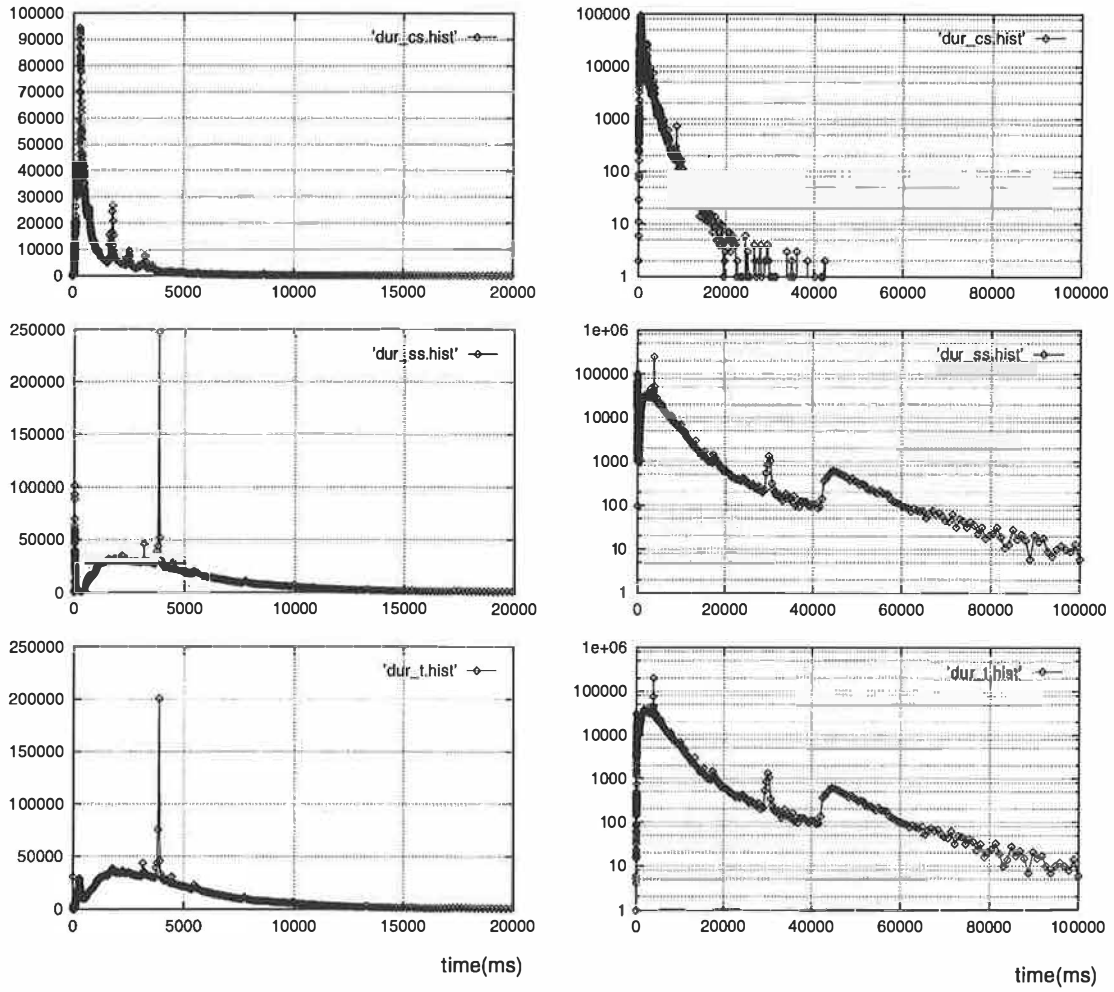


Figure 11: **Response time distributions** (a) elapsed time between the first observed byte from the client and the first observed byte from the server, (b) elapsed time between the first observed byte from the server and the last observed byte from the server, and (c) total elapsed time (between the first observed byte from the client and the last observed byte from the server). All distributions are shown with both a linear and a logarithmic Y-axis.

active users of a bank of 600-700 modems. Distillation is data-type specific, lossy compression - for example, a distilled image may have reduced resolution or color depth, sacrificing image quality for compactness of representation. Although a small additional latency is introduced by performing distillation, the byte-wise savings realized by the more compact distilled representations more than compensates for the latency of performing the distillation, resulting in a factor of 3-7 reduction in the end-to-end latency of delivering web content to users over their slow modem links. It was therefore an explicit design goal of TranSend to help mitigate the heterogeneity of Internet clients by adapting servers' content to clients' needs.

4.1 Burstiness

The TranSend service runs on a cluster of high performance workstations. Client requests are load balanced across machines in the cluster in order to maximize request throughput and minimize the end-to-end latency of each request through the system [15]. As observed in the Home IP traces, the load presented to TranSend is quite bursty on time scales on the order of seconds. Fortunately, the service time for an individual request is on the order of milliseconds; if a burst of traffic arrives at the system, it takes only a few seconds for the backlog associated with that burst to be cleared from the system.

Over longer time scales, we have indeed observed relatively stable, non-bursty load. Certain real-world events (such as the publication of an article in the campus newspaper about the service) did trigger temporary load bursts that persisted for hours, however these bursts were extremely rare (they have only occurred two or three times during the 4 months that TranSend has been active). Because of this long-term smoothness, we were able to allocate a fixed number of cluster nodes to TranSend. To handle the infrequent long-term bursts of activity, we designed TranSend to easily recruit "overflow nodes" in times of need.

4.2 Reference Locality

The TranSend service incorporates a large web cache. We have observed that there is locality in both the pre- and post- transformed representations of web content. In our experience, a 6 gigabyte web cache has been more than sufficient to serve the needs of the Home IP service, providing close to a 50% hit rate, as predicted by the cache simulations.

4.3 Service Response Times

The two largest components of end-to-end latency perceived by the end users of TranSend are the time that it takes TranSend to retrieve content from web services on a cache miss, and the time it takes TranSend to deliver transformed content to users over the slow modem lines. The time spent by TranSend actively transforming content is less than 100 milliseconds, but content retrieval and delivery latencies often exceed tens of seconds. This means that at any given time, there are many idle, outstanding tasks supported by TranSend, and a large amount of associated idle state.

We engineered TranSend to assign one thread to each outstanding task. Because of these high latencies, we have observed that there must be on the order of 400-600 task threads available. A large amount of the computational resources of TranSend is spent context switching among these threads. In retrospect, we concluded that a more efficient design approach would have been to use an event-driven architecture, although we would certainly lose the ease of implementation associated with the threaded implementation. Similarly, each task handled by TranSend consumes two TCP connections and two associated file descriptors (one for the incoming connection, and one for the connection within TranSend to the cache). We did not attempt to measure the overhead we incurred from this large amount of network state.

5 Related Work

A number of web client tracing efforts have been made in the past. One of the earliest was performed by Boston University [10], in which about a half million client requests were captured. These traces are unique in that the Mosaic browser was exclusively used by the client population; the Boston University researchers instrumented the browser source code in order to capture their traces. This research effort concentrated on analyzing various distributions in the traces, including document sizes, the popularity of documents, and the relationship between the two distributions. They used these measured distributions to make a number of recommendations to web cache designers.

Our traces are similar to the Boston University traces in spirit, although by using a packet snooper to gather the traces, we did not have to modify client software. Also, our traces were taken from a much larger and more active client population (8,000 users generating more than 24,000,000 requests over a 45 day period, as compared to the Boston University traces' 591 users generating 500,000 requests over a 6 month period).

In [20], a set of web proxy traces gathered for all external web requests from Digital Electronics Corporation (DEC) is presented. These traces were gathered by modifying DEC's two SQUID proxy caches. These traces represent over 24,000,000 requests gathered over a 24 day period. No analysis of these traces is given - only the traces themselves were made public. Only requests flowing through the SQUID proxy were captured in the traces - all web requests that flowed from DEC to external sites were captured, but there is a lack of DEC local requests in the traces.

Many papers have been written on the topic of web server and client trace analysis. In [32], removal policies for network caches of WWW documents are explored, based in part on simulations driven by traces gathered from the Computer Science department of Virginia Tech. In [9], WWW traffic self-similarity is demonstrated and in part explained through analysis of the Boston University web client traces. In [25], a series of proxy-cache experiments are run on a sophisticated proxy-cache simulation environment called SPA (Squid Proxy Analysis), using the DEC SQUID proxy traces to drive the simulation. A collection of proxy-level and packet-level traces are analyzed and presented in [12] to motivate a caching model in which updates to documents are transmitted instead of complete copies of modified documents. Finally, an empirical model of HTTP network traffic and a simulator called INSANE is developed in [23] based on HTTP packet traces captured using the *tcpdump* tool.

6 Conclusions

In this paper, we presented the results of an extensive, unintrusive client-side HTTP tracing efforts. These traces were gathered from a 10 Mb/s Ethernet over which traffic from 600 modems (used by more than 8,000 UC Berkeley Home IP users) flowed. Forty-five days worth of traces were gathered. We used a custom module written on top of the Internet Protocol Scanning Engine (IPSE) to perform on-the-fly traffic reconstruction, HTTP protocol parsing, and trace file generation. Being able to do this on the fly allowed us to write out only the information that interested us, giving us smaller and more manageable trace files.

We measured and observed a number of interesting properties in our Home IP HTTP traces, from which we have drawn a number of conclusions related to Internet middleware service design:

1. Although most web clients can be classified as accessing Internet services using a PC-based browsers and desktop machines, there is significant heterogeneity in the client population that Internet middleware services must be prepared to handle.

2. There is an extremely prominent diurnal cycle affecting the rate at which clients access services. Furthermore, clients' activity is relatively smooth at large time scales (on the order of tens of minutes, hours, or days), but increasingly bursty at smaller time scales (order of minutes or seconds). Internet middleware services can thus provision their resources based on the request rate observed over several hours if they can afford to smooth bursts observed over second-long time scales.

3. There is a very large amount of locality of reference within clients' requests. The amount of locality increases with the client population size, as does the working set of the client population. Thus, caches that take advantage of this locality must grow in size in parallel with the client population that they service in order to avoid thrashing.

4. Although Internet services tend to be very reactive, the latency of delivering data to clients is quite lengthy, implying that there could potentially be many hundreds or thousands of outstanding, parallel requests being handled by a middleware service. Services must thus minimize the amount of state and switching overhead associated with these outstanding, mostly idle tasks.

7 Acknowledgements

We would like to thank Armando Fox and Eric Anderson for their excellent feedback which improved the quality of this paper, and Clifford Frost for helping us to install our network tracing machine at the Berkeley Home-IP site. Thanks also go to Ian Goldberg and David Wagner for their helpful suggestions and guidance regarding the content of the traces, and to Vern Paxson for his help in publishing the traces in the Internet Traffic Archive.

References

[1] Allpen Software Home Page. http://www.allpen.com.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[3] Rob Barrett, Paul P. Maglio, and Daniel C. Kellem. How to personalize the web. In *Proceedings of the 1997 Conference on Human Factors in Compuer Systems (CHI 1997)*, Atlanta, Georgia, USA, March 1997.

[4] Jan Beran, Robert Sherman, Murad S. Taqqu, and Walter Willinger. Long-range dependence in variable-bit-rate video traffic. *IEEE Transactions on Communications*, 43:1566–79, March 1995.

[5] Tim Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol - http/1.0. RFC 1945, May 1996.

[6] Timothy C. Bickmore and Bill N. Schilit. Digestor: Device-independent access to the world wide web. In *Proceedings for the Sixth International World Wide Web Conference*, 1997. available at http://www.fxpal.xerox.com/papers/bic97/.

[7] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as http stream transducers. In *Proceedings of the 4th International World Wide Web Conference*, May 1996.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[9] Mark E. Crovella and Azer Bestavros. Explaining world wide web traffic self-similarity. Technical Report TR-95-015, Computer Science Department, Boston University, Oct 1995.

[10] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of www client-traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 1995.

[11] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of SIGCOMM '93*, September 1993.

[12] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the world wide web. In *Proceedings of the 1997 USENIX Symposium on Internet Technolgoies and Systems (USITS)*, Monterey, CA, USA, December 1997.

[13] Armando Fox and Eric A. Brewer. Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation. In *Proceedings of the 5th International World Wide Web Conference*, May 1996.

[14] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variation via on-demand dynamic distillation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1997.

[15] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Scalable cluster-based network services. In *To Appear in the Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-16)*, October 1997.

[16] Nicolas D. Georganas. Self-similar ("fractal") traffic in atm networks. In *Proceedings of the 2nd International Workshop on Advanced Teleservices and High-Speed Communications Architectures (IWACA '94)*, pages 1–7, Heidelberg, Germany, September 1994.

[17] Ian Goldberg. The internet protocol scanning engine. Personal communications.

[18] Graphic, Visualization, & Usability Center. 6th www user survey. Summary available at http://www.cc.gatech.edu/gvu/user_surveys/survey-10-1996/.

[19] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. In *Proceedings of the SIGOPS European Workshop*, September 1996.

[20] Tom M. Kroeger, Jeff Mogul, and Carlos Maltzahn. Digital's web proxy traces. Online at ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, aug 1996.

[21] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2, February 1994.

[22] Nikolai Likhanov, Boris Tsybakov, and Nicolas D. Georganas. Analysis of an atm buffer with self-similar ("fractal") input traffic. In *Proceedings of IEEE INFOCOM '95*, Boston, MA, April 1995. IEEE.

[23] Bruce Mah. An empirical model of http network traffic. In *Proceedings of INFOCOM '97*, Kobe, Japan, apr 1997.

[24] Benoit Mandelbrot. Self-similar error clusters in communication systems and the concept of conditional stationarity. *IEEE Transactions on Communication Technology*, COM-13, 1965.

[25] David Marwood and Brad Duska. Squid proxy analysis (spa). Report available at http://www.cs.ubc.ca/spider/marwood/, 1996.

[26] Microsoft Corporation Home Page. http://www.microsoft.com.

[27] Jeffrey C. Mogul. Operating systems support for busy internet servers. In *Proceedings HotOS-V*, Orcas Island, Washington, May 1995.

[28] Netscape Corporation Home Page. http://www.netscape.com.

[29] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Technical Conference*, June 1990.

[30] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of Poisson modeling. In *ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, London, UK, August 1994.

[31] Margo I. Selzter. Issues and challenges facing the world wide web. Talk presented to Lotus Corporation, available at http://www.eecs.harvard.edu/margo/slides/lotus.html, March 1997.

[32] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM*, Stanford, CA, USA, August 1997.

Alleviating the Latency and Bandwidth Problems in WWW Browsing

Tong Sau Loon Vaduvur Bharghavan
Department of Electrical and Computer Engineering
& Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{s-tong,bharghav}@crhc.uiuc.edu, http://timely.crhc.uiuc.edu/

Abstract

This work addresses three problems that are associated with Web browsing: (a) low bandwidth available to the end user who is connected via slow modems or outdoor wireless networks, (b) long and variable latencies in document access, and (c) temporary disconnections of mobile users. Three techniques are used with a variety of heuristics in order to overcome these problems: (a) profiling user and group access patterns and using these profiles in order to pre-fetch documents, (b) filtering HTTP requests and responses in order to reduce data transmission over bottleneck links, and (c) hoarding documents based on user profiles in order to support limited web browsing even during disconnection. In this paper, we describe the design and implementation of a WWW proxy-based system that incorporates the above techniques. We describe our experiences with the proxy system, and present performance results that show an improvement in the experience of Web browsing using this system.

1 Overview

In the last few years, the World Wide Web has had a remarkable effect on computing and communications in general, and Internet traffic in particular. Due to the explosion of the offered network load and the inherently best-effort paradigm of Internet service, WWW users typically notice long latencies and large variations in latency for web accesses. The scarcity of network bandwidth, particularly of the last link for users connected via low bandwidth modems and outdoor wireless networks exacerbates the latency problem, as does the transmission of increasingly graphics-oriented documents over slow networks. Mobile users face additional challenges in terms of frequent disconnections. In order to solve the above problems, we have built a WWW proxy-based distributed system which is compatible with existing browsers and protocol standards. This paper presents the design and implementation of our system and shows the performance improvements we obtained using this system in conjunction with our standard browsing environment.

Three problems motivate this work: (a) low bandwidth available to the end user who is connected via slow modems or wireless networks, (b) long and variable latency due to congestion in the network, best-effort service in the Internet, and transmission of large amounts of data over slow links, and (c) temporary disconnections of mobile users - either involuntary due to fades, or voluntary to save cost and battery power. In order to find effective solutions to the above problems, our work is based on three key observations: (a) User accesses to WWW documents have been shown to follow certain patterns, albeit changing over time [8] - these changing patterns can be learned by monitoring user accesses and used for both pre-fetching and hoarding. (b) Most large documents are graphics-intensive and graphics data can tolerate loss - thus, filtering images can significantly reduce data transmission without compromising severely on quality. (c) Groups of users with similar interests tend to access similar documents - the commonality of their access patterns can be learned by monitoring their access patterns. It should be noted that none of the above techniques are unique to our work. Caching of documents based on recent history of accesses is provided with most browsers (e.g. Netscape, Explorer). Intelligent pre-fetching based on document hyper-links and user access patterns have been proposed in several studies [3, 8, 9]. Filtering in order to adapt to dynamic network quality of service has been proposed in related work, both in the context of WWW accesses and in the context of application adaptation in general

[7, 16, 21]. Collaborative filtering has been proposed in the context of newsgroups [10] as well as WWW [1]. Hoarding has been proposed in the context of file systems support during disconnected operation [14]. The contribution of this paper is the combination of several mechanisms and the use of multiple heuristics in order to intelligently pre-fetch documents (based on user profiles, group profiles, and associated heuristics), filter documents (adaptively to varying QoS), and hoard documents anticipating disconnection (based on a hoard database that is learned over time as well as a user-defined hoard file). Performance results show that our system can learn and adapt to changing user behavior quickly, and significantly improve the experience of WWW browsing once the user profile is learned.

The rest of the paper is structured as follows. Section 2 describes the architecture of the system. Section 3 discusses the various heuristics employed in order to improve efficiency. Section 4 provides implementation details, while Section 5 presents performance results. Section 6 compares related work to our approach, and Section 7 concludes the paper.

2 Architecture of the WWW Proxy System

The three key aspects of our system design are pre-fetching documents based on user and group profiles, filtering retrieved documents based on the available network quality of service, and hoarding documents in anticipation of network disconnections (for mobile users). For pre-fetching and hoarding to be effective, the cached copy of the documents must be as close to the browser as possible. For filtering to be effective, it must be done as close to the server as possible; in particular, filtering needs to be performed *before* the bottleneck link on the retrieval path of the client, while pre-fetching and hoarding need to be done *after* the bottleneck link.

In the ideal case, a server would have a set of filters associated with a document type. A client request would be accompanied with the measured network quality of service. The server would then retrieve the document and pass it through the filter (with the QoS level as a parameter) before sending it back to the client. The advantage of this design is that only the required data is sent over the network, thereby decreasing the latency of access. Besides, if the user has to pay for receiving data over the network (proportional to the amount of data received), this mechanism can reduce the cost of Web access. The disadvantage of this design is that it requires QoS aware servers, and also places the burden of filtering on the server.

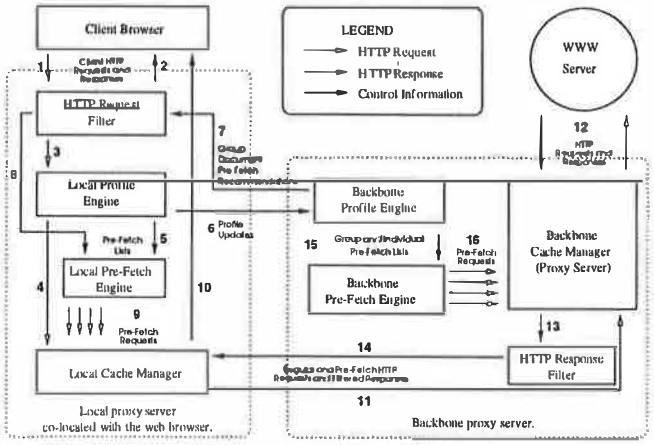


Figure 1: System Model

In cases where the user is connected to the network via a slow modem link or an outdoor wireless link, the last link typically happens to be the bottleneck link in the retrieval path. In this case, filtering the document before the last link may work just as well in reducing latency and maybe cost. Since this does not require any change in the server, we use this model for our system.

The architecture of our WWW proxy system is shown in Figure 1. A WWW browser points to a local proxy server, through which all requests are routed. The local proxy server contains an HTTP request filter, a profile management engine, a pre-fetching engine, and a cache manager. The local proxy server points to a backbone proxy server. Thus, the local proxy server acts as a server to the browser but as a client to the backbone proxy server.

The backbone proxy server essentially contains the same components as its local counterpart, but may service multiple users. The backbone proxy server thus handles both group profiles and individual profiles while the local proxy server handles only individual user profiles.

In order to effectively manage the usage profile, all user accesses must traverse through the local proxy server; thus, the browser's cache is disabled. This is because some HTTP requests would be intercepted by a browser cache if it were not disabled, and the local proxy server would not be able to learn the access pattern properly.

Profile-based pre-fetch is performed at both the local proxy server and the backbone proxy server, although the backbone proxy server does a more aggressive pre-fetch (more documents). Hoarding is done by the local proxy server. Filtering is done on both HTTP requests (e.g. to reduce HTTP headers) and HTTP responses (e.g. to clip images). The WWW server is not required to have any special functionality that is specific to our system.

2.1 HTTP Request/Response Paths

In this section, we provide an overview of each of the components in the architecture by means of an example of an HTTP request/response path. We then describe the functionality of each component in the system.

The numeric steps below refer the the steps shown in Figure 1:

1. When the user requests a document, the browser issues the request to the local proxy server.
2. This request first goes through an HTTP request filter. The request may be immediately satisfied (e.g. if the request is for a site that is blocked out, such as advertisements), may be modified (e.g. header compression [13]), or may be passed through without modification. Determination of filterable requests is based on substring matching of URLs to key strings and running corresponding scripts defined in a configuration file.
3. If a response was not generated immediately, the request is logged by the local profile manager and the user profile is updated.
4. The request is then passed on to the cache manager.
5. The profile manager will create a pre-fetch list based on the usage profile and send it to the pre-fetcher.
6. Requests which change the profile, specifically URLs which point to HTML pages, are sent to the backbone profile engine to enable backbone aggressive pre-fetches and to update the backbone profiles. Note that the use of an explicit connection to send the profile updates is mainly for ease of implementation. A more efficient mechanism would be to piggy-back such data on HTTP requests that gets transmitted from the local proxy server to the backbone proxy server.
7. Periodically, the backbone profile engine returns a list of recommended pages to pre-fetch based on group profiles. This can occur when many users of a particular group visit a particular page. Similar to above, such information can be piggy-backed onto HTTP responses in a more efficient implementation.
8. The recommended URLs are operated on by a function in the HTTP request filter to eliminate URLs that would be filtered (i.e., we do not want to pre-fetch items that we will filter). This new list is submitted to the pre-fetcher.
9. The pre-fetcher collates the pre-fetch list and group document pre-fetch recommendations that are found to be not filterable. It then amortizes the pre-fetch requests to the cache manager.
10. If the cache has a fresh copy of the document originally requested, the request is satisfied immediately.
11. Otherwise, the request is forwarded to the backbone proxy server.
12. The normal HTTP transaction occurs between the backbone cache manager and the WWW server.
13. After retrieval, the document is passed through the backbone HTTP response filter.
14. The response is sent back to the local cache manager, who will cache the document if it is a cacheable item. It is then sent back to the browser (10).
15. The backbone profile manager maintains individual as well as group profiles. Periodically, it creates a list of recommended group documents and sends it to the local proxy server (7) of each member of the group. As profile updates arrive, it creates a list of documents to pre-fetch based on individual and group usage profiles. The only difference from the local pre-fetch list is that the backbone list is longer (i.e., we do more aggressive pre-fetches on the backbone). This list is then submitted to the backbone pre-fetch engine.

16. The backbone pre-fetcher will issue the necessary pre-fetch requests.

2.2 HTTP Request/Response Filters

The filter engines are responsible for modification of HTTP requests and responses in order to reduce data traffic over the bottleneck link. Typical examples of filtering include reducing the color depth of images, clipping images, transmitting parts of audio files only, transmitting the first few words of each paragraph in a document, reducing the HTTP request header, suppressing requests to documents from a particular server, etc. Every user has his/her own HTTP request filter and HTTP response filter.

Invocation of the appropriate filter is accomplished by means of a rules database. A rule in the database specifies the invocation of a filter based on its document type, available network quality of service, size of the document, etc. The goal of the rules database is to allow adaptive filtering based on dynamic network conditions and data types. While the set of rules in the rules database is currently small, the architecture allows for more sophisticated filtering.

The HTTP request filter also notifies the user of the documents that are pre-fetched via a well-known URL. That is, it intercepts a well-known URL and formulates an HTML page which represents the pre-fetched items in the cache. This allows the user to see which URLs have been pre-fetched and manually set or modify the list.

2.3 Usage Pattern Profile

A usage profile is a representation of a user's or group's usage pattern of the Web. The profile is learned over time by monitoring the stream of HTTP requests of users. Using the profile to determine which documents to pre-fetch, rather than simply using the document layout, can significantly improve the efficiency of pre-fetching. The usage profile is a directed weighted graph, where the nodes represent URLs and edges represent the access path. The weight of a node u indicates the frequency of access of the corresponding URL, while the weight of an edge *(u,v)* indicates the frequency of access of the URL v immediately following the URL u. In order to reflect the changing access patterns of users and the temporal locality of accesses, recent history is given precedence in the weighting heuristic.

Let $n_t(u)$ represent the number of accesses of node u (i.e. the URL corresponding to node u) during the time interval t, $n_t(u,v)$ represent the number of accesses of node v immediately following node u, $w_t(u)$ represent the weight of node u after the time interval t, and $w_t(u,v)$ represent the weight of edge (u,v) after the time interval t. The weights of nodes and edges are computed as follows:

- $w_{t+1}(u) = w_t(u).\alpha_1 + n_t(u).\beta_1$
- $w_{t+1}(u,v) = w_t(u,v).\alpha_2 + n_t(u,v).\beta_2$

where α_1, α_2, β_1, and β_2 are constants which indicate the relative weights of recent history versus past history. These constants, along with the time window t, play a key role in determining how effectively the profile adapts to the changing user access patterns. Modifying these parameters will determine whether the profile does long term adaptation or short term adaptation. Based on our own experience, we have currently set $\alpha_1 = 0.9, \beta_1 = 0.1, \alpha_2 = 1.0$, and $\beta_2 = 1.0$. This means that we have set a very high weightage to previous history. This is because we wanted our system to be insensitive to spurous bursts of visits to sites that will not be visited again i.e. long term adaptation. We have set t to be the time between two successive *sessions*. This means that the weights are recalculated at the beginning of every Web session. While the current values of the constants are based on what worked for our usage profiles, we plan to do more experiments in order to determine the weights in the graph. While our weights are determined solely by frequency of access, we plan to use the following parameters for determining weights in the future: percentage of membership that uses an edge or node for group profiles, expected latency, liveliness of documents, and size.

Group usage profiles are a natural extension of the idea of exploiting individual usage profiles to predictively pre-fetch documents. They also fit naturally into collaborative filtering of documents discussed in [10]. Group usage profiles are inherited by users that join the group. Thus, a new user would first enroll in several groups. This will ensure that there is some form of informed speculative pre-fetch service for the user while his/her own usage pattern is being learned by the local profile engine. In addition, as the interests of the group change over time, the user will be able to inherit the changes automatically.

In the case of group profiles, we keep more state. In particular, we are interested in the number of members in a particular group who have visited a particular URL and who have used a particular edge.

This allows us to make recommendations on which URLs to pre-fetch. For example, if more than 50% of the group uses a particular edge (u, v), then we recommend v when a member of the group visits u.

While individual profiles are used for predictive pre-fetches, group profiles serve two purposes. First, a group profile is inherited by a new member to a group, hence, while his/her individual profile is being learnt, it is still possible to predictively pre-fetch. Second, group profiles are useful for notifying a member of pages that most of the other members of the group are visiting.

2.4 Profile Engines

The local profile engine is mainly responsible for maintaining a single user's usage profile. It receives filtered HTTP requests from the HTTP request filter and then updates the profile. It then creates a pre-fetch list based on the profile and submits the list to the local pre-fetcher. At the same time, the local profile engine updates the backbone profile engine.

The backbone profile engine is responsible for maintaining individuals' profiles as well as group profiles. After it receives an update from the local profile engine, it creates a (longer) list of items to pre-fetch based on the individual's profile and submits it to the backbone pre-fetcher. Note that the list based on individual profile is longer because we want to ensure that items are readily available at the backbone proxy server, should it not be found in the local cache. It also creates another pre-fetch list based on the groups that the user has subscribed to. This list is then submitted to the backbone pre-fetcher. Periodically, a list of recommendations are sent back to the local proxy server. Note that instead of pushing the document directly to the user, we employ the use of a notification system with the local proxy server making the final decision of whether to pre-fetch a document or not. This pull with notification mechanism is more flexible than a pure document push from the backbone proxy server because the user might not need all the documents.

2.5 Pre-fetch Engines

The local pre-fetch engine's main responsibility is to issue pre-fetch requests to the local cache manager based on the lists obtained from the local and backbone profile engines. The recommendation list that arrives from the backbone profile manager goes through the local filter first, in case there are some documents that the individual wishes to block out. It then amortizes the pre-fetch requests over time so that the local cache manager does not get overloaded. At the same time, the local pre-fetch engine makes sure that duplicate items are not pre-fetched.

A special case of pre-fetch is *hoarding*, which is done only by the local pre-fetch engine. In the case of hoarding, only the node weights are used in order to pre-fetch the documents that the user is most likely to require in the future. As in the case of disconnected file systems, this gives rise to the problem of caching versus hoarding. An approach similar to hoard-walking [14] is used in order to refresh commonly used items.

It should be noted that on the local machine, user requests are given priority over pre-fetch requests. That is, no pre-fetch requests are issued until there are no pending user requests.

The backbone pre-fetch engine's main responsibility is to issue pre-fetch requests based on the list obtained from the backbone profile engine. Note that multiple lists can be submitted to the backbone pre-fetcher if there are multiple users logged on. The backbone pre-fetcher will ensure that duplicate items are not pre-fetched and that pre-fetch requests are amortized over time.

2.6 Cache Manager

Each proxy server has a cache manager which maintains a cache of documents available as a result of user requests, pre-fetches, and hoard-walks. When the proxy server receives a request, if the document is not available at the cache, the request is forwarded to the next level of proxy or directly to the WWW server. Note that the browser cache is disabled, since all user requests must go through to the local proxy server in order to build an accurate user profile.

Should disconnections arise, the local cache manager is also responsible for providing the documents. In our system, if any control connection with the backbone machine is broken prematurely, the local proxy server will go into a *disconnected* mode. In this mode, it will only present to the browser the items on the local cache. If a request is made for an item that was not hoarded, the local cache manager returns an empty file. The browser will then present a "Document contains no data" message to the user.

The backbone cache manager is any standard off-the-shelf proxy server. In the current implementation, we use *Squid* [4].

3 Performance Enhancing Heuristics

We used several heuristics in order to improve the efficiency of pre-fetching. While many of the heuristics listed below are simple and intuitively obvious, we found that a combination of these heuristics provided a remarkable performance improvement in our daily operation. This section lists the heuristics we used.

1. *Web Sessions*: The notion of a Web session had one of the greatest performance impacts. Without the notion of a session, the pre-fetch engine re-issued multiple requests for the same documents if a user accessed the same pages again later in the session. With the notion of a session, documents are only fetched once during the session. Subsequent requests are satisfied from the cache, unless the user explicitly requests a fresh access using the RELOAD button (which issues a HTTP "Pragma: no-cache" header). At the start of a session, documents with the highest node weights are hoarded. The idea of Web sessions is not new. Current browsers all have a notion of a Web session.

2. *Hoard walking*: Hoard-walking [14] periodically refreshes pages with the highest node weight. Since hoard-walking involves pre-fetching the pages in the user defined hoard file as well as the most heavy nodes in the learnt database, consequently, a large fraction of typical user accesses can be satisfied locally during disconnection.

3. *Issue Of Pre-Fetch Requests*: Pre-fetches are performed only when the network is "idle". In our system, only four ongoing pre-fetches are allowed at any time at the local proxy server and only eight ongoing pre-fetches are allowed at any one time in the backbone proxy server. Furthermore, on the local proxy server, pre-fetches are not started until there are no pending user requests. We observed performance improvements when we amortize pre-fetch requests. Of course, not issuing pre-fetch requests while there is a pending request might actually *lower* performance, especially in the case of requests with high delays. However, giving priority to user requests eliminates the harmful effects of pre-fetch tying up bandwidth when it is needed.

4. *Weighting edges*: Using weighted edges as opposed to only using node weights for pre-fetches ensures that the proper usage pattern is captured. When a user visits a URL, we should choose the edge with the heaviest weight rather than the adjacent node with the heaviest node weight. For example, consider the following scenario: C was visited 2 times from A and 100 times from B. B was visited 10 times from A. When a user is at A, B should be pre-fetched even though it has a smaller node weight than C.

5. *Dynamically determining a document's dependents*: We distinguish a document from the images that are linked to it (which constitute its dependents). Dependents do not appear in the user profile, because they are accessed automatically upon access of the document, and also because they change frequently over time. The original implementation stored the URLs of the dependents. However, due to the frequent changes in HTML documents, requests were being made for non-existent documents. This heuristic removed all those redundant pre-fetch requests, at the same time keeping the user profile graph small. Furthermore, pre-fetch requests for dependents are issued (at both the local proxy server and the backbone proxy server) before the browser issues them.

6. *Continued download of document*: Even after the user specifies "stop", we continue downloading a document in the local proxy server. This allows a user to click on another page while the previous page is being downloaded in the background. It also serves as a crude form of short-term user-specified hoarding or "user-driven pre-fetch". Even though this is not captured in the performance tests, we found it extremely useful when we were reading a page with links we knew we wanted to visit. We would click on the link and quickly press stop. This would issue the pre-fetch request but keep the browser on the current page. Later, when we eventually clicked on the page, it came up instantly.

7. *CGI scripts*: CGI and other dynamic pages are pre-fetched and retained in the cache for a short period of time. Currently, CGI pages are not cached either in browsers or proxies; thus

CGI latency is not hidden. However, with this heuristic, we can pre-fetch CGI pages and cache them for short periods of time in anticipation of an access. A CGI page is deleted upon the timeout, or after it is read once.

8. *HTTP Redirections*: HTTP response codes 301 and 302 indicate that a particular URL has moved. If the local proxy system detects this, it will store the redirection and provide the correct response to the browser. This prevents the browser from reconnecting to the server.

9. *Thresholds*: Since many documents are only accessed once from a parent document (e.g. a news item from a topic), we impose a minimum threshold edge frequency in order to pre-fetch the node. Another threshold we impose is the size of documents. Large documents (more than 1 megabyte) are not pre-fetched in our current implementation. However, we anticipate that with the inclusion of size as a weight parameter, this heuristic will be subsumed in the weighted edge heuristic.

4 Implementation

The WWW proxy caching system is written in C++ in the Linux environment. On the local machine, the user needs to run only one process, the local proxy server called *localCache*, in addition to the browser itself. On the backbone server, several processes need to be run. These are the backbone cache manager *squid* [4], the HTTP response filter *filterd*, the backbone pre-fetch engine *pfetcher*, the backbone group profile manager *gpm*, and the backbone communication surrogate *bcs* which talks to *localCache*. We will describe these processes in more detail next. It must be noted that other than *squid* [4], all the other processes can be combined into a single one that spawns into different modules. The use of different processes is purely to ease debugging. In essence, *filterd* and *bcs* keep states about the current session. The rest of the processes keep state about the backbone proxy server.

4.1 Local Proxy Server

The local proxy server, *localCache*, implements all the different parts of the local proxy server shown in Figure 1. Specifically, the HTTP request filter is implemented as a method call that does substring matching on HTTP requests and invokes scripts to create HTTP responses based on the rules database. The local profile engine is implemented as a class (a directed graph with weighted edges) which trims itself when the number of nodes grows too large. Currently, it trims itself when the number of nodes reaches 1024. It does so by removing half of the nodes (512) that are lightest according to the weighting method we described earlier. The local pre-fetch engine is implemented using a queue. The local profile engine feeds the queue with URLs. When a pre-fetch request is issued, the URL is removed from the queue and placed into a session set. The same URL will then no longer be pre-fetched, even if it was placed in the queue again. The local cache manager is implemented as a class that interfaces a URL to a disk file via a hashing function. Currently, HTTP response headers are also stored. In addition to disk store, we also keep a copy of pre-fetched items in virtual memory. This is more for ease of implementation than for performance. An optimized version of the program can use just the disk store and thus leave a small memory map.

Of particular interest with the local cache manager is the determination of an item's expiry time. Let `t_expire`, `t_date`, and `t_last_modified` be times (in seconds) extracted from the respective HTTP response headers. If a header is not available, the variable defaults to 0. Further, let `HTML_expire` and `IMAGE_expire` be the times (in seconds) the user provides. These times specify how long the user wants an HTML file or an image file to stay fresh in the absence of some/all of the above HTTP response headers. Let `t_now` be the current time in seconds at the local machine and `expire` be the time when the document expires. The algorithm for determining the freshness of an item in the local cache manager is as follows:

```
if (t_expire>0 && t_date>0)
  expire = t_expire - t_date + t_now;
else if (t_last_modified>0 && t_date>0)
  expire = (t_date - t_last-modified)/2 + t_now;
else if (t_last_modified>0)
  expire = (t_now - t_last_modified)/2 + t_now;
else if (HTMLDocument(url))
  expire = HTML_expire + t_now;
else
  expire = IMAGE_expire + t_now;
```

4.2 Backbone Proxy Server

For the backbone cache manager, we use *squid* [4]. However, any proxy server that understands standard HTTP/1.0 should work, since all communication with the backbone cache manager occurs via HTTP/1.0 requests.

The HTTP response filter is implemented in the process *filterd*. *filterd* reads a configuration file upon being started by a running copy of *bcs*. The configuration file indicates what scripts to run on what kinds of responses. For example, JPEG files might require the script *jpegfilter*. *filterd* listens to a well known port for incoming HTTP requests. It then forwards all requests to *squid* [4]. After it receives the response from *squid* [4], it checks to see if the response is filterable. In the above example, if the response file is called *picture.jpg*, *filterd* will issue the command:

```
jpegfilter picture.jpg tempfile.nam QoS
```

The script is expected to produce the filtered response in *tempfile.nam* based on the QoS parameter. *filterd* then makes necessary changes to the HTTP response headers and sends the new response back. Basically, *filterd* maintains the state of pending HTTP requests, responses, and the connection to the local cache manager and *squid* [4].

The QoS parameter measures the quality of service of the network connections between the local proxy server and the backbone proxy server along two somewhat orthogonal dimensions: rate and delay. Periodically, the local proxy server measures the HTTP request-response round trip time (RTT) for a request and signals the backbone proxy server to do the same for the same request. (This is done via piggy-backed HTTP headers.) The difference in the measured RTTs at the local and backbone proxy servers gives an RTT estimate for the HTTP request. By fitting a curve over a sequence of such measurements, the local proxy server can estimate the rate and delay QoS parameter. These values are then transmitted to *filterd* and fed back into the local HTTP request filter.

The backbone pre-fetch engine *pfetcher* keeps state of all current Web sessions. It is able to do this because every copy of *bcs* connects to it. It also maintains a queue of URLs that it feeds (as HTTP requests) to *squid* [4]. Each running copy of *bcs* will send their own pre-fetch list. *pfetcher* will amortize these pre-fetch requests to *squid* [4]. As pre-fetch responses arrive, *pfetcher* will determine the dependents and issue more pre-fetch requests if necessary.

The backbone group profile manager *gpm* keeps track of all the groups which have their group profiles stored on that particular backbone server. It also maintains the list of members of each group. It acts as a query-response server to the group profiles database. When a running copy of *bcs* informs *gpm* where its user is currently located, *gpm* will update the group profiles of all the groups that the user is subscribed to. It then returns a list of recommendations based on the updated group profiles to *bcs*.

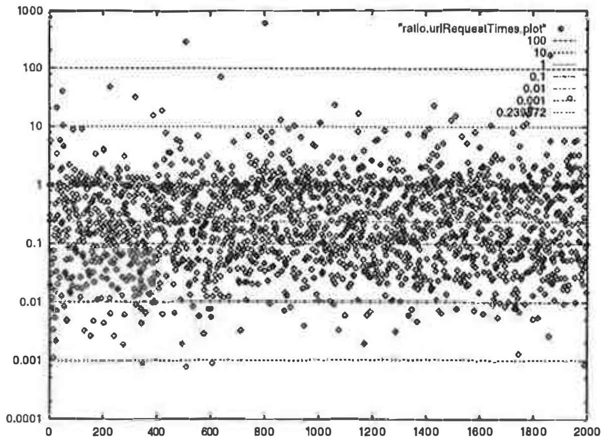


Figure 2: Ratio of individual URL request times. X-axis represents each of the 2000 URLs. Y-axis represents the ratio of delay of each URL with the proxy system to the delay with only *squid* [4].

The backbone communication surrogate *bcs* is a forking process that forks itself for every new Web session by a different user. It maintains the per-user states. That is, it maintains the user profile, the groups a user belongs to, and the user's current QoS. It updates *gpm* about the movements of the user and obtains a pre-fetch list from it when it wants to recommend pages to the user. It connects to *pfetcher* and sends it a list of items to pre-fetch based on the user's individual profile. This is for the aggressive backbone pre-fetch. It also spawns the user's HTTP response filter, i.e. *filterd*.

5 Performance Measurements

Performance of the system was evaluated with a browser simulator called *surf*. *surf* reads a text file containing a list of URLs to access. It then issues HTTP requests for each of the URLs and fetches the dependents if necessary. *surf* simulates user reading time by sleeping an amount of time proportional to the HTML file size without the HTML tags. This is called the *readspeed*. If a URL is not retrieved within a certain amount of time, the whole URL is abandoned. This is specified by the *impatience* parameter and is used to detect Web sites that are down. There is also the caching version of *surf* called *csurf*. The only difference is that *csurf* fetches every unique URL only once per session. This simulates the internal browser cache.

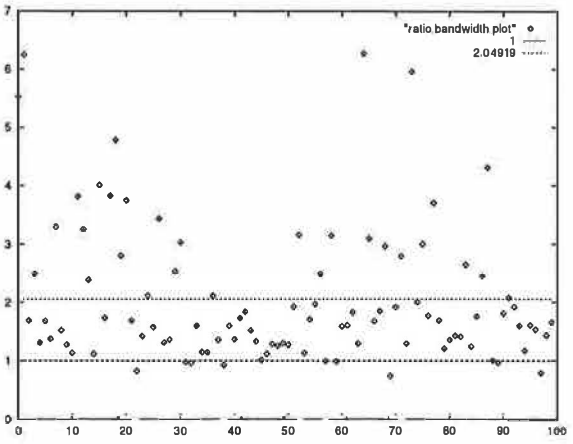


Figure 3: Ratio of session HTTP bandwidth. X-axis represents each of the 100 sessions. Y-axis represents the ratio of the HTTP bandwidth of the session with the proxy system to the HTTP bandwidth of the session without the proxy system.

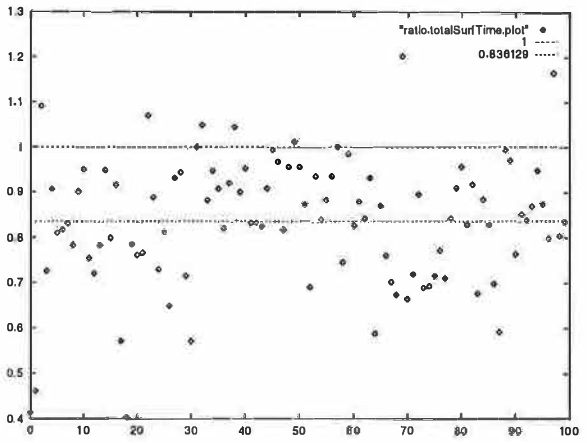


Figure 6: Ratio of total surfing time. X-axis represents the 100 sessions. Y-axis represents the ratio of the total time of the session with the proxy system to the total time of the session without the proxy system.

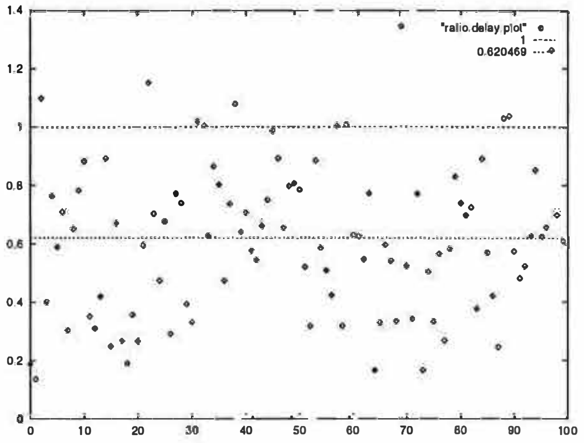


Figure 4: Ratio of average HTTP request delay. X-axis represents each of the 100 sessions. Y-axis represents the ratio of the average request delay of the session with the proxy system to the average request delay of the session without the proxy system.

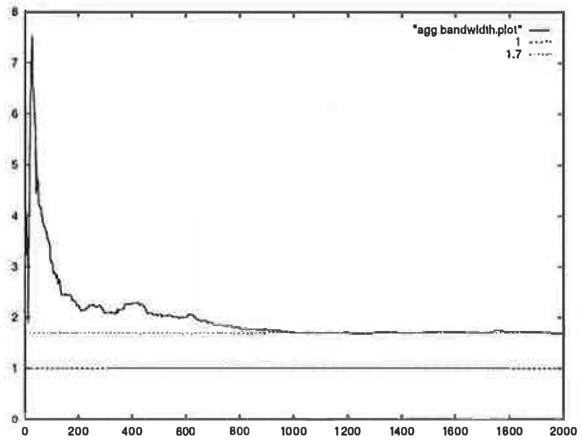


Figure 7: Ratio of HTTP bandwidths over 2000 URLs. X-axis represents the 2000 URLs. Y-axis represents the ratios of the HTTP bandwidth for all the previous URLs.

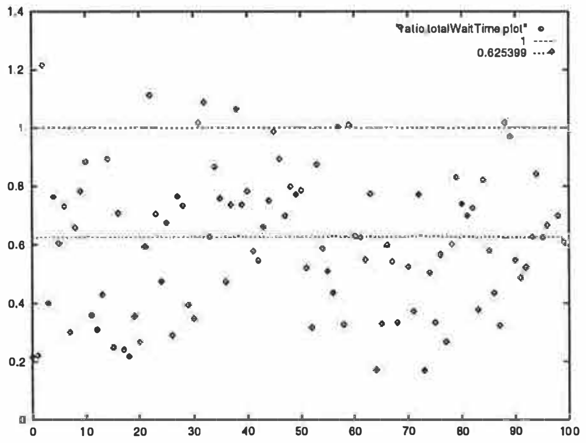


Figure 5: Ratio of total network wait time. X-axis represents the 100 sessions. Y-axis represents the ratio of the total network waiting time of the session with the proxy system to the total network waiting time of the session without the proxy system.

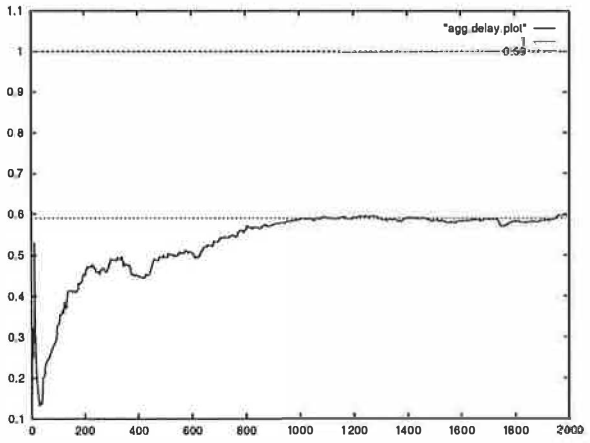


Figure 8: Ratio of average HTTP request delay over 2000 URLs. X-axis represents the 2000 URLs. Y-axis represents the ratios of the average HTTP request delay for all the previous URLs.

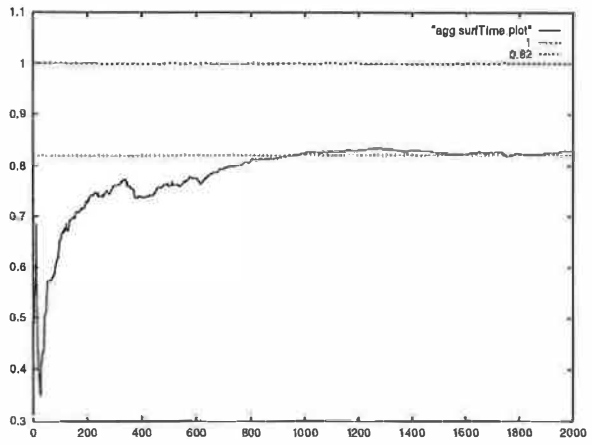


Figure 9: Ratio of Web session times. X-axis represents the 2000 URLs. Y-axis represents the ratios of the total session time for all the previous URLs.

We obtained 2000 random URLs from the Yahoo! Web site and divided them into 100 groups of 20 URLs to represent 100 different sessions. We then ran the *csurf* on the URLs with *squid* [4]. On a different client-server system, we ran *surf* with the proxy system (before the test, we ran surf 5 times for each of the 100 sessions over the proxy system to allow time for the proxy system to learn the access pattern). For both the tests, we set *readspeed* to be 128 chars/second and *impatience* to be 60 seconds. We synchronized the sessions between the two tests so that it cancels out the network activity that is time-based. We also cleared all caches (local and backbone) between sessions.

Figure 2 shows the ratio of each URL's delay time (HTML file together with image file times) plotted on a log scale. 79% of the URLs have a ratio of less than 1. This means that 79% of the URLs took less time with the proxy system. The 21% of the URLs that took equal or longer time can be attributed to the network variance in the Internet. The median ratio was 0.239572.

Figure 3 shows the ratio of each session's HTTP bandwidth. We see that on the average, we get twice the bandwidth, as far as the browser is concerned. Note that this is the mean value.

Figure 4 shows the ratio of the average URL request time per session. We see that on the average, each request takes only about 62% of the time it used to take. This means a faster response to the browser.

Figure 5 shows the ratio of the total network waiting time of each session. We notice that with the proxy system, on the average, we only wait 62.5% of the time we would normally wait in one session. Note that this graph is similar to Figure 4. It is only similar and not exact because of dynamic pages that return different dependents each time e.g. advertisements.

Figure 6 shows the ratios of the total time taken to complete each of the 100 sessions. The times included network waiting time, sleeping time (to simulate reading), and CPU overhead time. We see that even though 21% of the individual URL request times were equal or slower, grouping URLs into sessions amortizes the time, and only 8%-10% of the sessions are slower than before. We also see that on the average (over 100 sessions), we spend only 83% of the time we spend on a Web session when we are using the proxy system.

Finally, we collated the times of the 2000 URLs to see how the proxy system will perform in the long term. It must be noted that the value we get here is actually lower than what we would have gotten had we used a 2000 URL session. This is because we cleared all caches between sessions.

Figure 7 shows that, on the average, the browser observes a bandwidth increase of 1.7 times when the proxy system is being used.

Figure 8 shows that, on the average, for individual HTTP request delay, the browser waits only 59% of the time. As expected, this is roughly the inverse of the bandwidth improvement.

Figure 9 shows that, on the average, the user spends only 82% of the time he/she would have if he/she only used a single backbone proxy.

5.1 Caveats

Pre-fetching is pointless if the user can read at a rate faster than data can arrive at his/her browser. We assume that this is not the case.

In general, we have observed that the profiles are learned for the first time over 5 to 7 sessions. Learning changes in user patterns online is dependent on the weights assigned. In our case, we placed heavy emphasis on a user's history. This means it takes longer for the system to adapt to changes. At the same time, the system is less sensitive to random bursts of visits to certain sites that will not be visited again. That is, we set the system to do long term adaptation. We found that this model suited our access patterns.

The cache hit ratio, defined as the number of hits over the number of user requests, for the above tests averaged at 62%. The accuracy of pre-fetches, defined as the number of hits over the number of pre-fetches made, averaged at 50%. However, note that *surf* does not visit previous pages. In normal browsing, we have found that the "previous" button is used rather often, and this increases the hit ratio as well as the pre-fetch accuracy. Note that the pre-fetch accuracy can actually be greater than one if a single pre-fetch can service multiple requests. In our own browsing, we found that the hit ratio hovers at around 75% and the pre-fetch accuracy hovers at around 70%. In this case, the low pre-fetch accuracy is due to visits of new sites in daily Web surfing.

Note that all these numbers are specific to our access patterns and for our learned profiles. We expect that these numbers may change for other usage patterns - in particular, the tuning of the constants in determining user profiles played a key role in improving the efficiency of pre-fetching. However, while we believe that significant work needs to be done in order to automatically tune the system to match user access patterns, we do believe that our system can provide perceptible improvements in the experience of Web browsing.

We also noted with interest that our own access patterns have changed as a result of using the proxy system. However, we felt that a user's access pattern will naturally change when the network condition changes. For example, if the network is slow, the user is usually apprehensive about clicking and then waiting. Since our proxy system basically changes the network conditions as far as the browser is concerned, we felt that a change in our access pattern is tolerable.

6 Related Work

Studies on techniques which aim to reduce the latency of Web accesses include LowLat [19] and WebExpress [13]. LowLat differs from our system in that it requires a process to be located near the Web server. WebExpress differs in that it uses file caching, forms differencing, protocol reduction, and the elimination of redundant HTTP header transmission to reduce the bandwidth used. Furthermore, WebExpress multiplexes multiple HTTP requests over a single link to reduce the TCP setup overhead. Currently, our system transmits HTTP documents through standard HTTP/1.0.

Studies into speculative pre-fetch of Web documents include work done by the OCEAN group [6, 9, 8], ICS-FORTH [15], Tenet [17, 18], and Wcol [5]. OCEAN's approach differs in that they use both server initiated pre-fetch as well as client-initiated pre-fetch. Further, they use a Random Walk User Model and a DSP User Model to model usage patterns. ICS-FORTH differs in that they employ a server initiated pre-fetch with the help of a Top-10 Approach. Tenet represents usage pattern on the server through dependency graphs. Similar to our pre-fetch with notification, their server makes the predictions and the client initiates the pre-fetches. Wcol differs from our profile-based pre-fetch in that they parse the HTML files and pre-fetch both the links and the inline images. Wachsberg [20] describes the use of a model similar to ours. A commercial product that does speculative pre-fetch is PeakJet [3].

Studies on geographical push caching [11, 12] by the VINO research group involves server initiated pushing and differs from our client initiated approach.

Studies into collaborative data filtering include Tapestry [10], and FIREFLY [1]. JunkBusters [2] is a proxy server that also filters HTTP requests. Our work is similar to the architecture that Zenel describes for intelligent filtering in low-bandwidth environment in that we make use of an intermediary (proxy).

7 Summary

Users surf the WWW in a regular fashion; thus, it is possible to exploit that information in caching systems. Furthermore, since users spend a non-trivial amount of time reading a page, the time can be used to pre-fetch documents rather than let the network stay idle. This paper described the use of usage profiling, pre-fetching, and filtering techniques in the context of WWW caching.

The usage profiles employed the use of a directed graph to represent the path a user takes in surfing the web. This information is later used by the pre-fetch engine to issue pre-fetch requests to the cache manager. Meanwhile, the HTTP requests and responses are filtered to ensure good use of the available bandwidth.

Using various heuristics described in the paper, we implemented a proxy system that improved the network performance from the perspective of the browser. This had the effect of reducing the overall time spent on web sessions.

References

[1] FireFly. http://www.firefly.com/.

[2] Junkbusters. http://www.junkbusters.com.

[3] PeakJet. http://www.peak-media.com/.

[4] Squid Internet Object Cache. http://squid.nlanr.net/Squid/.

[5] WWW Collector - The Prefetching Proxy Server for WWW. http://shika.aist-nara.ac.jp/products/wcol/wcol.html.

[6] Azer Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. *Proceedings of CIKM'95: The 4th ACM International Conference on Information and Knowledge Management*, Nov 1995.

[7] Vaduvur Bharghavan and V. Gupta. A Framework for Application Adaptation in Mobile Computing Environments. *Proceedings of the IEEE Computer Software and Applications Conference, Washington D.C.*, Aug 1997.

[8] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-Based Traces. *Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215*, Apr 1995.

[9] Carlos R. Cunha and Carlos F. B. Jaccoud. Determining WWW User's Next Access and Its Application to Pre-Fetching. *Proceedings of ISCC'97: The Second IEEE Symposium on Computers and Communications*, Jul 1997. (Extended version).

[10] D. Goldberg, D. Nichols, B.M. Oki, and D. Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM, Volume 35, Number 12*, Dec 1992.

[11] James Gwertzman and Margo Seltzer. An Analysis of Geographical Push-Caching. http://www.eecs.harvard.edu/vino/web/server.cache/icdcs.ps.

[12] James Gwertzman and Margo Seltzer. The Case for Geographical Push-Caching. *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, May 1995.

[13] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. *MobiCom'96 Demonstration Session*, Nov 1996. http://www.networking.ibm.com/art/artwewp.htm.

[14] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the CODA File System. *ACM Transactions on Computer Systems, Vol. 10, No. 1*, Feb 1992.

[15] Evangelos P. Markatos and Catherine E. Chronaki. A Top-10 Approach to Prefetching on the Web. *Technical Report No. 173, ICS-FORTH, Heraklion, Crete, Greece.*, Aug 1996. http://www.ics.forth.gr/proj/arch-vlsi/www.html.

[16] B.D. Noble and M. Satyanarayanan et al. Agile Application-Aware Adaptation for Mobility. *Proceedings of the ACM Symposium on Operating Systems Principles, St. Malo, France*, Oct 1997.

[17] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving World Wide Web Latency. *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, Jul 1994.

[18] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *ACM SIGCOMM Computer Communication Review*, Jul 1996.

[19] Joe Touch. The LowLat Project. http://www.isi.edu/lowlat/, 1996.

[20] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast World-Wide Web Browsing over Low-Bandwidth Links. http://ccnga.uwaterloo.ca/~ sbwachsb/paper.html, 1996.

[21] Bruce Zenel and Dan Duchamp. Intelligent Communication Filtering for Limited Bandwidth Environments. *Proceedings of the fifth Workshop on Hot Topics in Operating Systems Rosario, Washington*, May 1995.

The Search Broker

http://sb.CS.Arizona.EDU/sb/

Udi Manber   Peter A. Bigot

Department of Computer Science
The University of Arizona
Tucson, AZ 85721-0077
{udi,pab}@CS.Arizona.EDU

Abstract

The current search facilities on the web are amazingly powerful, but they are still lacking. Taking the whole universe as one flat data space and searching it with keywords has inherent limitations of scale. The challenge is to provide users with ways to focus their search better without making it too difficult or too inefficient. We introduce a method of conducting search on the web that is based on a two-level search idea. It strikes a balance between flat global search and specialized databases, and gives users convenient access to vast amounts of information.

1 Introduction

Large scale web search started in two directions, both of which proved very successful. The first is "spider"-based collection of as much of the web as possible, combined with powerful search engines that provide access to all this data. Lycos, WebCrawler, Altavista, and several others are examples of this model. The second is based on manual collection and classification with browsing and search facilities. Yahoo is, of course, the most successful provider of this model. This is not the place for a detailed analysis of these two approaches; let us just mention their most obvious weaknesses: The spider-based approach's main problem (and often main strength as well) is that it is indiscriminatory. It tries to cover everything. It is not uncommon to obtain thousands of hits, most of which are "garbage," and then have to sift through many of them. It's also impossible to expect to have all the world's information in one flat database. All spider-based engines take only a small fraction of most large sites. (E.g., you cannot expect them to collect all 16GB of Medline, and as a result they will not give you all pertinent medical information.) Yahoo's approach guarantees more quality, but browsing is often very time consuming, and of course, coverage is limited.

These weaknesses are most glaring when one looks for answers to specific reference questions, such as:

- how much fat is there in a pepperoni pizza?
- how do you say *search* in Latin?
- how do you delete a directory in UNIX?
- give me a list of hotels in Phoenix.

These are hard questions to answer based on keywords alone and flat search. Using the spider-based services, one will have to think of the right keywords, and if they are too common, a lot of hits will have to be followed with no guarantee of quality of information. Yahoo will probably be more suitable for these questions, not by trying to answer them directly, but by trying to find the right categories (e.g., dictionaries, although Yahoo doesn't know about a Latin one), then following them to hopefully the right places that maintain relevant information. But in any case, it could be quite time consuming (e.g., try to find the list of Phoenix hotels).

The approach we present here is based on the idea of a two-level search. Instead of always searching the same all-encompassing database, imagine having specific databases for specific topics. The search will consist of two phases: In the first phase, the search is after the right database, and in the second phase the relevant information is searched for within this database. This is not a new approach, of course. It is similar, in a sense, to using the library subject card catalog to find the right shelf, or using Yahoo to find the right category. The novelty of our tool is that we combine the two phases into one regular search in a way that makes the process very easy and very powerful for users. The resulting tool provides search features that are not available in any one place on the web.

Consider again the three questions listed above. The first one has to do with nutrition, the second with Latin, the third with UNIX, and the fourth with hotels. These are the most important characteristics of these questions. The person who asks the question can usually pinpoint its subject; not precisely, maybe, but usually close. For example, the questions above could be answered more precisely if they were in the following forms:

- Subject: nutrition; Query: pizza
- Subject: latin; Query: search
- Subject: unix; Query: delete directory
- Subject: hotels; Query: Phoenix

Knowing the right subject may be tricky. Some users may input calories instead of nutrition, or accommodations instead of hotels. We only ask that some information about the subject be included in the query. We also replace the rather complex syntax above with a very simple query, as we will show shortly.

Our approach works as follows. We collected over 400 different search providers that we judged to have reasonable general appeal. (This is an on-going process, of course; we expect a fully-operational system to have thousands of servers.) Each such search server covers a certain subject or category (such as nutrition, latin, unix, or hotels). Each such category is identified by one or two words, and it is also associated with a list of *aliases* that people may think about when searching for that subject. So nutrition can be associated with calories, and hotels with motels, lodging, and accommodations. The collection of search engines and the assignment of the words and aliases that identify them are done manually by a librarian. It can also be customized by end-users (we'll discuss that aspect later on). This is a part that we intentionally do not wish to automate. The role of editors, reviewers, interpreters, and librarians has been rather limited in the web, mainly because of its scale. Finding paradigms that will allow significant librarian input while supporting the scale of the web is increasingly important. The two-level approach is promising because the number of subjects does not grow too fast (as compared to the number of web pages or even the number of web sites).

A user query is made of two parts corresponding to the two phases of the search. In the current implementation both parts are combined into one simple box. To answer the questions above you type:

- nutrition pizza
- latin search
- unix delete directory
- hotel Phoenix

and you get direct results from the appropriate search services. Given a query like hotel phoenix, the Search Broker performs the following steps:

1. It searches its own database for subjects and aliases and finds the search engine corresponding to hotel. With aliases, all subjects allow both plural and singular names (hotels works as well as hotel). In the current implementation, the subject must be the first word in the query, mainly because we want users to identify the subject and think about it. We could easily select any word in the query that matches a subject and try it (or all of them).
2. After identifying the particular search engine, the rest of the query is reformatted to the form expected by that search engine. This step can sometimes be quite complicated, and we discuss it in detail later.
3. An HTTP (Hypertext Transfer Protocol [Fielding]) request is sent to the search engine with the appropriate fields that match the query.
4. The results of the query are sent back to the user.

This simple-minded approach turns out to be extremely powerful. The proliferation of search software, often for free, made it easy to provide search capabilities on many web sites. (We are proud to be partially responsible for that with our glimpse, glimpseHTTP, WebGlimpse, and Harvest systems.) Within the last year thousands of search servers have been added. Most of them deal with very limited specific information (e.g., they search the content of one site), but many provide professional content in areas of general interest. The trend to connect existing databases to the web will continue. There are already so many high quality search facilities that people cannot keep track of them through bookmarks and favorite lists.

The list of currently available subjects is included in the home page of the Search Broker, and there are also facilities to search the Search Broker's own database. Let's see some examples of queries to demonstrate the power of the approach:

stocks ibm gives the current value of IBM's stock plus links to corporate information and news.

patent object oriented gives abstracts of all patents (from 1971 to present) with these keywords.

howto buy a car gives practical advice about buying used and new cars.

fly sfo jfk gives all scheduled flights between San Francisco and New York JFK.

english-polish wonderful tells you that *cudowny*, *zadziwiajacy*, and *godny podziwu* match the adjective "wonderful".

nba-salaries michael gives the salary of Michael Curry (and all other Michaels playing in the NBA).

car-price 1992 Chevrolet, Camero gives the blue book value for this model (the comma (,) is used as a delimiter).

email bill gates gives email addresses for that name (yes, it includes the one you are thinking of).

travel fiji gives a lot of useful information about travel in Fiji.

expert computer algorithm gives a list of experts who put "computer algorithms" in their areas of specialty.

convert 8 liters to pints tells you that there are 16.9 pints in 8 liters.

The Search Broker approach is not a magic bullet, and we do not expect it to replace any of the existing search mechanisms. But we believe that it complements them very well. If one is not sure what one is looking for, browsing works best, and Yahoo presents the right approach. If one is looking for unusual words or names or wants everything known about something, then the spider-based engines cannot be beat. But queries often fall somewhere in between, and the Search Broker can help save time and focus the results. In a sense, it presents a very large live encyclopedia.

The first version of the Search Broker has been operational since October 1996. It was released on the web in July 1997. It can be found at `http://sb.cs.arizona.edu/sb/`.

The rest of the paper is organized as follows. The next section presents the user interface, always an important part of any tool. Then we discuss technical details of how we create and maintain the database of search engines, handle queries, and return results to the user. After that we present a different and maybe even more important use of the Search Broker—as a customizable desktop tool. Related work and conclusions follow.

2 The User Interface

We have experimented with several user interfaces. Our primary interface follows the minimalist approach. It is just one generic search box with a Submit and Reset buttons as shown in Figure 1.

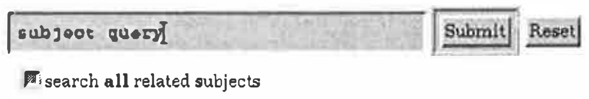
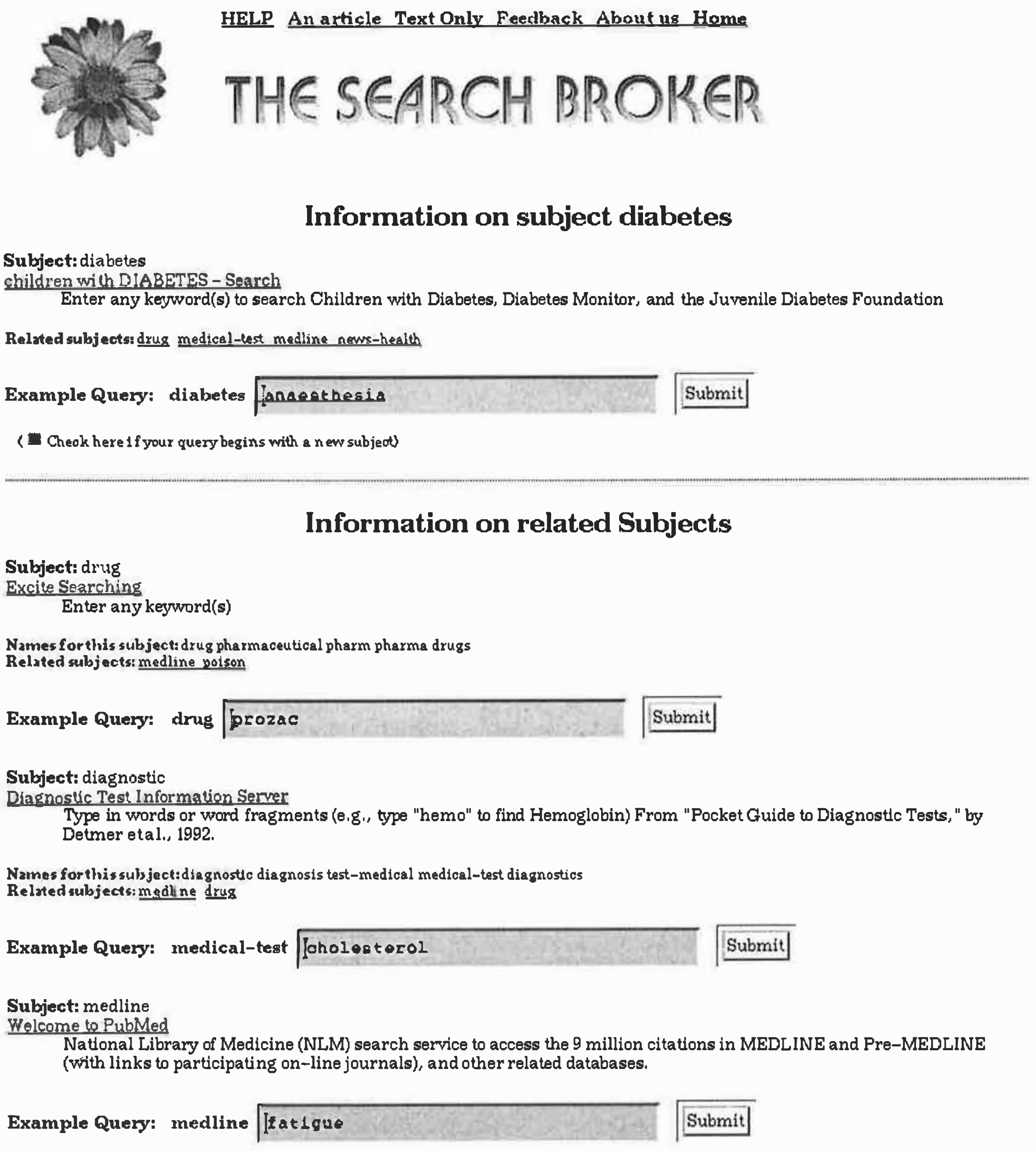


Figure 1: The minimalist user query box

The convention is that the first word is the subject and the rest is the actual query. If the subject requires more than just one word, the words are joined with hyphens; for example, book-kids, tv-guide, or programming-languages. Sometimes the second part is not needed; for example, tv is also an alias for tv-guide, but book is a different subject covering all books rather than just books for kids.

The main problem is how to let users know which subjects exist and what they cover. This is achieved in two ways. First, the list of subjects (divided into several "super" subjects, such as computers, entertainment, and science) is provided below the search form on the Search Broker home page. Second, we provide an extensive help system to search for subjects. For each subject we maintain a list of *related subjects*, again decided manually. If the query contains only one word, then this word is searched for in the list of subjects and information about it and all its related subjects is given. For example, typing *diabetes* will bring the content shown in Figure 2.

If there is no subject matching the first word, the user is presented with the options of forwarding the query to another search engine (using a Search Broker subject like lycos), or revising the query and resubmitting it. Future enhancements may include proposing alternative subjects based on words close to what the user entered (spelling corrections), or by using a more extensive topical-relation database. It is also possible to employ natural language processing techniques to try to automatically infer the subject from the query. We have not tried that yet. Another possibility is the use of pull-



Information on subject diabetes

Subject: diabetes
children with DIABETES - Search
Enter any keyword(s) to search Children with Diabetes, Diabetes Monitor, and the Juvenile Diabetes Foundation

Related subjects: drug medical-test medline news-health

Example Query: diabetes anaesthesia Submit

(Check here if your query begins with a new subject)

Information on related Subjects

Subject: drug
Excite Searching
Enter any keyword(s)

Names for this subject: drug pharmaceutical pharm pharma drugs
Related subjects: medline poison

Example Query: drug prozac Submit

Subject: diagnostic
Diagnostic Test Information Server
Type in words or word fragments (e.g., type "hemo" to find Hemoglobin) From "Pocket Guide to Diagnostic Tests," by Detmer et al., 1992.

Names for this subject: diagnostic diagnosis test-medical medical-test diagnostics
Related subjects: medline drug

Example Query: medical-test cholesterol Submit

Subject: medline
Welcome to PubMed
National Library of Medicine (NLM) search service to access the 9 million citations in MEDLINE and Pre-MEDLINE (with links to participating on-line journals), and other related databases.

Example Query: medline fatigue Submit

Figure 2: The results of diabetes

down menus to show the existing subjects so that users do not have to guess them. 400 subjects in one pull-down menu is out of the question, but they could be divided into several categories. From initial experiments, this option does not look attractive to us.

Another feature is an option to search all related subjects with one search. For example, if this option is selected, then the query dictionary bravo (or word bravo) will also search the thesaurus, jargon, phrase, and quotation subjects at the same time.

We place a Search Broker form at the start of the results pages returned from the remote server, so users can make additional queries without having to move to a different page. The initial user interface required the user to have the subject word at the beginning of all queries. By watching how people used the Search Broker, it became clear that users automatically invoked another pattern, where they used the subject in an initial query, but left it off in subsequent ones, assuming searches would continue with the same first-level restriction. To support this usage, we modified the query form on result pages so that the subject would remain the same, and users enter only the second level of the query, unless the user specifically indicated that she wanted a new two-level search.

3 The Search Broker's Database Facilities

The database for the Search Broker contains all the information required to locate an engine that can service a user's query, to rearrange and submit the query in the format expected by the engine, and to provide brief descriptions and examples for each search. This information includes:

- The method and action of the query form (e.g., `GET` and `http://search.yahoo.com/bin/search`);
- A list of form inputs, including their types (text, hidden, checkbox), initial states, and valid states where appropriate (radio and select types);
- A subject name and all its aliases, and a list of related subjects;
- Instructions on how to assign field values from the user's input (query templates; cf. section 4);
- Examples, documentation, and a reference the user can follow back to the original search engine for more refined searching or additional information about the site.

The current database occupies only about 300K with more than 400 subjects.

The accuracy of much of this information is crucial for correct processing of a user's query. For a tool like the Search Broker to be successful, its database must be constantly verified and updated, adding new search engines and removing or modifying entries for ones that have disappeared or changed. We developed a tool which automates much of the tedious portions of database creation and maintenance. Given a URL of a search page, our tool automatically retrieves the HTML search form, translates the action and input fields into a concise format from which the important parts of the original form can be reconstructed, and outputs a database entry template. The librarian then assigns a subject name and aliases, writes a description of the search engine's capabilities, selects which of the form fields to use and sets default values for the rest, and provides query translation patterns. The entry is then ready to add to the database.

In our experience, the "administrative" parts of a subject can typically be added to the database in 20-30 seconds, and therefore do not present a significant burden. The main job of the librarian is to find the right search facilities, give them the most appropriate names and aliases, test them for quality and generality, and write good short descriptions and examples for them. This is in line with the traditional responsibilities and expertise of librarians, which in our opinion the web sorely misses. The Search Broker's design allows librarians to make significant contributions in organizing the web's search facilities.

The same program can also be used for routine maintenance, by querying the entire database on its own. It will go out and verify that each server listed is still there, and that the search form it uses has not changed. If something has changed—the form has moved, or has new fields—the differences are noted for the librarian to review before changing the database.

We are currently extending this program to allow for personal customized use so that people can easily be their own librarians and maintain their own Search Brokers. More on than in the Customization section.

4 Translating Queries

The problem of reformatting queries between different databases is an old problem in the database area. Fortunately, the web makes it simpler, because queries typically use few fields and they typically allow search of the whole database by (non attributed) keywords. HTML forms are often very simple. We built a pattern-matching based scheme to translate the Search Broker queries to different HTML forms.

The Search Broker's database query template consists of a list of form inputs, an extended regular expression against which the user's query is matched, and an optional block of Perl code used to post-process the assignments and set dynamic defaults. The majority of forms have only one field that is filled by the user's query; for example, a text input for keywords. Such a query is defined in the database very simply:

```
keywords = (.*)
```

Currently, about 90% of the subjects use this simple template. Parentheses are used to group regular expressions into a single value that is assigned to an input field. One can have many different fields. For example, some forms require separate fields for city and state. The query template we use in this case is:

```
city state = (.*),\s*(.*)
```

which translates "Tucson, AZ" into "city=Tucson state=AZ" and sends these fields to the search engine.

Sometimes the search form contains several required fields, a few of which are not essential to many queries. For example, in the flight schedule form, there are fields for departure time, departure day, departure month, favorite airline, etc. We picked only the departure and arrival cities as the two necessary fields and use the database query post-processing feature to assign defaults (e.g., two weeks from now, all airlines) to the rest of the fields. We believe that the ability to send a super-simple query

```
fly sfo jfk
```

outweighs the weakness of not including all the details. Of course, those extra fields can be used directly on the original form which the user can obtain from our help system. In a few cases we used several subjects for the same form with different defaults; for example, cd-by-artist and cd-by-title.

Each database entry can have multiple query templates. The templates are matched against the user's input in their database entry order, and the first one that matches the input is chosen. This allows us to support a variety of input formats, rather than force the user to guess which one we can recognize. For example, a server that provides a list of famous people born on a certain date might accept only one format of the date (e.g., in HTTP format: `mon=06&amp;day=25&amp;year=1953`), but templates could be provided to recognize and reformat all of the following:

```
famous-births 6/25/53
famous-births 25 June 1953
famous-births June 25th, 1953
```

The number of recognized formats is limited primarily by the imagination and energy of the librarian.

Because HTML forms often restrict field values through the use of radio or select input types, some search engines are very finicky about the format of queries they accept (for example, requiring the leading zero in the month input example above, something a user would not normally provide). This combination of extended regular expression pattern matching and arbitrary post-processing code has proven to be a very powerful solution to the problem of reformatting queries to meet these requirements.

5 Formatting Responses

After user input has been converted into an HTTP GET or POST request, the Search Broker contacts the remote server and retrieves the response. In the common case, we simply append the body of the response to an introduction that includes a description of and reference to the source search engine, and a form for additional queries. To avoid any appearance of claiming the content of responses as our own, we generally do not modify the retrieved material—inline graphics, scripts, and advertisements are left in their original place.

There are two cases where we must modify the returned material. The first is due to the latitude of HTML, which allows closing tags to be omitted, with an implicit close at the end of the document or entity. When we return results from multiple servers in response to a user's "search all related servers" request, we must append closing tags to the end of each response so that font changes, table layout, and other formatting directives do not affect the subsequent responses.

Modification is also used when a server is known to provide far more information than we want. An example of this is the flag subject. There is no search form interface to this information; the URL points to a long list of links to GIF-format images of flags. The Search Broker database entry provides a response post-processing feature, through which the retrieved material is split into sections based on some pattern (for example, end of line, start of HREF), and only the sections which match the user's query are returned. Although the current database format does not allow for arbitrary response post-processing as it does for query input reformatting, it can easily be extended to support well-defined actions such as filtering by pattern matching, or cutting out particular regions of a response.

6 Customization

So far we discussed the idea of the Search Broker in the context of one central server. But the same method can be applied to personal use, and it can lead to very powerful customizable search facilities on anyone's desktop. The current database, which includes over 400 subjects each with instructions, source, and examples of its use, together with the whole Search Broker software occupy less than half a megabyte. They make use of CGI scripts and assume an HTTP server, but there is no reason they cannot be used on any desktop through direct interaction with the browsers (e.g., on UNIX through Remote Control of UNIX Netscape, or on Windows through ActiveX).

Imagine having a search box somewhere on your desktop to which you can assign your own names to whichever subjects you choose, which sends your simple queries (e.g., fly sfo jfk) directly to the appropriate place, and presents the results automatically on your (favorite) browser. On the one hand, it can be thought of as a more convenient personal "hot list" of search facilities. You don't have to store (and know about) all the different search facilities; you don't have to load the selected search engine's own interface; you don't have to figure out how to fill out the selected form; all you need to do is use your own aliases (or try the standard ones). But it can be even more.

You could customize subjects such as "my-directions" which will give road directions to any address from *your* home. You could search local information. For example, you could have a subject called bookmarks which searches in your own bookmark list using aliases you give to the different pages or any word in their title or URL; you could have a subject called history which searches in the list of all URLs you've ever seen; you could have a subject called file which acts as a "Find File" application, searching for a file name on your local disk; you could have a birthday category searching your own lists of birthdays, a calendar searching your schedule, and so on. You could link some of these subjects so that a search for a person's name will be first conducted on your address book, then on your whole file system, then on your organization's database, then on the whole web. All of these searches could also be done at the same time, giving you the results in the right order. This can become your own personal oracle.

We are currently building a new Java-based version of the Search Broker to allow users to easily set up search scripts, whereever and whichever search engine they want to use.

7 Related Work

The seed of the Search Broker grew out of the Harvest project.[Bowman *et al.*], where we attempted something similar, but ended up concentrating on the actual collection of data rather than the selection of servers. Harvest is an integrated system to collect, extract, organize, index, search, cache, and replicate information across the Internet. It is used by hundreds of sites to build "brokers" (our reuse of this term here is coincidental and unrelated) which serve collections of information gathered from many sources. Harvest has a special broker called the Harvest Server Registry (HSR), which maintains information about all brokers. The original intent was that there would be enough Harvest brokers to be used for most purposes, and that the HSR will lead people (by queries) to the right broker. That never happened, and Harvest never extended this idea to facilitate easy selection of servers.

The closest existing search facilities to the Search Broker are the lists of search engines, such as The Internet Sleuth and C/Net Search.com. We believe that our approach is an improvement, and has the potential, especially with personal customization, to be a significant step forward. The Search Broker is easier to use, and it does not require downloading or browsing large pages with forms. IBM InfoMarket also provides source selection through pull-down menus, limited to several publications that offer pay-per-view.

There has been a lot of work in the Information Retrieval area to support natural language queries (e.g., [Salton]). The search engine attempts to "understand" the essence of the query, to figure out which words are more important and which could be substituted more effectively, and how to assign different weights based not only on the query but also on the database. The success of these methods has been mixed. They can lead to unusual findings or embarrassing misses. In the context of the web, where the information is as diverse as possible and as unstructured as possible, it is very difficult to infer structure and patterns. We believe that our approach of handling the selection of search servers and assignment of subjects to them by hand is both feasible and desirable.

There has been a lot of research in the database community related to source selection and interaction between separate databases. Wiederhold [Wiederhold] introduced the general notion of *mediators*. Levy *et al.* [Levy1, Levy2] developed tools (e.g., the Information Manifold) to allow complex queries across different databases. Their most pressing problem is how to negotiate with complex database schemas, a problem we don't have (yet).

The idea of a *MetaCrawler* to combine results from several sources was introduced by Selberg and Etzioni [Selberg]. We currently just concatenate results if they are obtained from several sources. Incorporating this technology would certainly help.

There has also been research in the expert systems area in source selection. For example, the Reference Expert from the University of Houston [Bailey] was developed to help users select the right reference material in the library for their questions, and QPEA (Query Planning Environment Assistant) [Huffman] is being developed at Price Waterhouse to help specialists select and combine data sources.

8 Conclusions

The Search Broker offers a balance of focused search, ease of use, and generality. It opens the door to a more significant involvement of experts in the organization of search. It explores the middle ground between completely automated search systems on the one hand and manual collection of information on the other. The web searching problem is too big a problem to be solved by one tool or even one model. The Search Broker presents a slightly different model than the existing ones, and for some users and some purposes it gives excellent results with much less effort than other approaches. We strongly believe that the Search Broker model can capture an important niche and encourage people to make available more specialized search facilities, which will benefit everyone.

9 Acknowledgements

This work was supported in part by NSF grant CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052. The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

References

[Bowman *et al.*] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz, `<URL:ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Conf.ps.Z>`. The Harvest Information Discovery and Access System, *Computer Networks and ISDN Systems* **28** (1995) pp. 119-125. (An early version appeared in the *Proceedings of the Second International World Wide Web Conference*, pp. 763-771, Chicago, Illinois, October 1994.)

[Salton] Gerard Salton, *Automatic Text Processing*, Addison Wesley, Reading, Mass, 1989.

[Wiederhold] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *IEEE Computer*, **25** (1992), pp. 38-49.

[Levy1] Alon Y. Levy, Anand Rajaraman and Joann J. Ordille, `<URL:http://www.research.att.com/~levy/vldb96-im.ps.Z>`. Querying Heterogeneous Information Sources Using Source Descriptions, *Proceedings of the 22nd International Conference on Very Large Databases*, VLDB-96, Bombay, India, September, 1996

[Levy2] A.Y. Levy and J.J. Ordille, `<URL:http://cm.bell-labs.com/cm/cs/doc/95/11-01.ps.gz>`. An Experiment in Integrating Internet Information Sources, *AAAI Fall Symposium on AI Applications in Knowledge Navigation and Retrieval*, Cambridge, MA (November 1995).

[Selberg] Erik Selberg, and Oren Etzioni, `<URL:http://www.cs.washington.edu/research/projects/softbots/papers/metacrawler/www4/html/Overview.html>`. Multi-Service Search and Comparison Using the MetaCrawler, Proceedings of the 4th International World Wide Web Conference, 1995.

[Bailey] Charles W. Bailey Jr. and Robin N. Downes, `<URL:http://educom.edu/stories.101/Intelligent-Reference.txt>`. Intelligent Reference Information System (IRIS), in *101 Success Stories of Information Technology in Higher*

Education: The Joe Wyatt Challenge, edited by Judith Boettcher, McGraw-Hill, New York, 1993.

[Huffman] Scott B. Huffman and David Steier, "A Navigation Assistant for Data Source Selection and Integration," in *Working Notes of AAAI-95 Fall Symposium Series on AI Applications in Knowledge Navigation and Retrieval*, pp. 72-77, Cambridge, MA, 1995.

[Fielding] Fielding, *et al.* Hypertext Transfer Protocol – HTTP/1.1. Network Working Group Request for Comments 2068.

Using the Structure of HTML Documents to Improve Retrieval

Michal Cutler, Yungming Shih, Weiyi Meng
Department of Computer Science
State University of New York at Binghamton
Binghamton, NY 13902
{cutler, meng}@binghamton.edu

Abstract

The World Wide Web (WWW) is a gigantic information resource, which is growing daily. As more and more data are added to the WWW, it is becoming increasingly difficult to effectively locate useful information from this environment. In this paper, we propose a method for making use of the structures and hyperlinks of HTML documents to improve the effectiveness of retrieving HTML documents. Our study assigns the occurrences of terms in a document collection into six classes according to the tags in which a particular term appears (such as Title, H1-H6, and Anchor). Based on the assignment, we extend the weighting schemes in traditional information retrieval by incorporating different importance factors to terms in different classes. The rationale is that terms appearing in different places of a document may have different significance in identifying the document. For this research we have built a Web based search tool, Webor, created a testbed, and conducted extensive experiments to determine an optimal class importance factor combination. Our study indicates that substantial improvement of retrieval effectiveness can be achieved using this technique.

1. Introduction

The World Wide Web has become an important information resource today. The popularity of the Web is primarily due to the tremendous amount of information available, and the ability to browse and publish information. At the same time, as more and more data are added to the WWW every day, it has become increasingly difficult to effectively locate useful information from this environment.

There are typically two different approaches for finding information in the WWW. The first is *browsing*. Browsing based systems organize information in the WWW into category hierarchies. Examples of this type of systems are Yahoo [Yaho96] and Magellan [Mage97]. The second approach is *searching*. When using a searching based system (called a search tool), a user submits a query and the system returns a list of web pages (usually the URLs of these pages) that are potentially useful to the user. Many search tools have been employed and some examples are Alta Vista [Alta96], WebCrawler [Pi94], Lycos [CMU95, Mau97], OpenText [OTC96], WISE [YuLe96a, YuLe96b] and Microsoft Index Server [Mi97]. Most browsing based systems also provide a more general searching capability.

A search tool is essentially a Web-based information retrieval system. Such a system typically consists of two components. One is a robot-based *indexing engine* which recursively downloads web pages, indexes the contents of downloaded documents, extracts URLs from them and downloads more web pages using these URLs. In the end, an index database organized as an inverted file is constructed. The second component is a *search engine*, which compares each user query to the downloaded pages through the index database and returns a list of web pages, which are potentially useful to the user. Usually, the returned web pages are ranked based on some similarity function.

The work presented in this paper is based on the vector space model [Salt83]. In traditional vector space based information retrieval, each document is represented as a vector ($w_1, w_2, ..., w_k$), where k is the number of distinct terms in all documents in the system (usually after *stopwords* such as `a' and `of' have been removed and a *stemming* algorithm has been applied to convert words to their stems), and w_i is a real number indicating the *weight* (or significance) of the *i*th term in identifying the document. If a term does not appear in a document, then its corresponding weight in the document vector is zero. The weight of a term *t* in document *d* is usually computed from two factors. The first factor is the *term occurrence frequency*, *tf*, which is the number of times *t* appears in *d*. Intuitively, a larger *tf* should imply a larger weight, so the weight of term *t* should be proportional to *tf*. The second factor is the *document frequency*, *df*, which is the number of documents in the collection that contain *t*. Intuitively, the more documents that contain *t*, the less significant the term is in differentiating *d* from other documents. Therefore, a larger *df* should imply a smaller term weight. In other words, the weight of term *t* should be proportional to the *inverse document frequency*, *idf*, of *t*. A commonly used formula for computing the weight of *t* in *d* is $tf \cdot idf$.

In the vector space information retrieval model [Salt83, Salt89], each user query is also represented as a vector ($q_1, q_2, ..., q_k$), where q_i is the weight associated with the *i*th query term. Usually, q_i is either zero, indicating that the query does not contain the *i*th term, or one, indicating that the *i*th term is in the query. Sometimes $tf \cdot idf$ is also used for computing q_i. The closeness (or similarity) of a query and a document is often computed based on the inner product of their vectors. Similarities are often normalized to between 0 and 1 through the use of a normalization factor.

In this paper, we study extending traditional approaches for information retrieval to the WWW environment. There are two key differences between the documents in this environment and those used in traditional IR systems.

1. The structure of HTML documents is easily available through HTML tags. For example, in an HTML document, we can easily tell whether a term appears in the title, one of the six headings or whether it is emphasized by using an underscore, italics or bold characters. Such structures provide information about the content of a document. Intuitively, terms that appear in the title, header, or are emphasized in the text are more important for retrieval than the rest of the terms. So by storing the structure information of HTML documents in the index and assigning an appropriate importance value to the appearance of the terms in each structure, the structure and the importance information can be used to improve the rank assigned to retrieved documents.

   Traditional IR systems typically disregard information about the structure of a document. The main reason for this is that commonly this information is either not available, or is hard to acquire.

   It is well known that a search can be improved by taking the structure of a document into account. Several search engines already use tag information to improve ranking. AltaVista [Alta96], HotBot [HotBot], and Yahoo [Yaho96] score a document higher if query words or phrases are found in the title of a web page. Lycos [Mau97] uses position information on query term occurrence (title, body, header, one of 100 most relevant words) in the rank function. The insight into why some structure information is used while other structure information is ignored is not published. In addition, the weights assigned to the various structures in the similarity function, and the information on how these weights were derived are not available. This paper presents a systematic investigation of HTML structure information, explains how the importance assigned to tagged terms was derived, and how the similarity function was modified.

2. HTML collections contain additional information about each document *d* that has hyperlinks to it in other documents of the collection. Typically, when authors add a hyperlink to a document *d*, they include in the Anchor tag, a description of *d* in addition to its URL. These descriptions have the potential of being very important for retrieval since they include the perception of these authors about the contents of *d*. In particular, they may provide good synonymous and related terms which are not included in the text of *d*, but may be included in user queries. So by adding the anchor information of HTML documents to the index we may be able to retrieve documents that could not be retrieved otherwise. In addition, using an optimal importance value for anchor terms may improve the rank assigned to retrieved documents.

   In traditional IR approach, only terms included in a document are used to automatically index it (unless a thesaurus is used). There have been several approaches to combining hypertext with information retrieval [Agost96]. For retrieval from a hypertext medical handbook, Frisse took into account the occurrence of query terms in the descendants of a hypertext document [Frisse88]. Dunlop [Dunlop93] used the cluster of documents citing and cited by a document for retrieval. Retrieval from hypertext was also investigated by Croft [Croft93], and Frei [Frei92].

Research was also conducted on how to take advantage of the additional information available in the hierarchical (or graph) structure of Web collections to improve retrieval. Yuwondo and Lee [YuLe96a, YuLe96b], considered a number of alternative ranking algorithms. The algorithms are based on the idea that neighbors of a web page are related to the page and that this information can be used to improve ranking. *Boolean spread activation* increased the belief in a document if query terms appeared in its neighbors. *Most-cited* increased the belief in a document if query terms appeared in its parent pages. *Vector spread activation* increased the belief in a document depending on the similarity of its children to the query. Hypursuit [Hypur96] combined document similarity with hyperlink semantic similarity. The hyperlink semantic similarity is based on two documents having a path of links connecting them, the number of ancestor documents that refer to both, and the number of descendant documents that both documents refer to.

In this paper, we propose a systematic method for extending retrieval techniques to include HTML structures. We first group subsets of HTML tags into a set of classes. Then when we index each document *d*, we assign the occurrence of each term *t* of *d* into one of these classes. In addition, the occurrences of term *t* in the anchor structure of other documents that have hyperlinks to *d* are assigned to an additional class, called the Anchor class. Next, we attempt to learn an optimal importance factor combination for the classes, by conducting extensive retrieval experiments. Finally, the best importance factor combination found is used to adjust the weights of terms and to compute the similarity of queries to documents. The proposed method provides us with the necessary flexibility to test the effectiveness of making use of HTML tags and hyperlinks to enhance retrieval. To carry out the experiments, we created a testbed with a collection of 4,596 HTML documents and ten queries. For each query, the set of relevant documents and the set of irrelevant documents were identified manually.

This paper has the following contributions. First, a testbed was created. The availability of such a testbed is essential to carry out performance studies in information retrieval. While there are standard testbeds for traditional information retrieval (for example the TREC collections), there is currently no standard testbed for web documents. Creating a testbed requires a lot of effort as relevant and irrelevant documents for each query need to be identified manually. Second, a systematic method was proposed for studying the effectiveness of using HTML tags and hyperlinks to enhance document retrieval in the WWW environment. Third, our experiments indicate that by storing structure information in the index, assigning optimal class importance values, and using the extended index and importance values, retrieval effectiveness is substantially improved.

The rest of the paper is organized as follows. In Section 2, we present Webor, which is the tool we built [Webor96] and used for this study. The classes used by Webor, its *indexing engine*, and its *search engine* are also briefly described. In Section 3, the testbed is described. More specifically, we discuss the document collection and the queries of the testbed, as well as the methodology for determining the sets of relevant and irrelevant documents for each query. In Section 4, we describes the experiments, the results obtained, and how the best importance factor combination is determined. Section 5 provides some conclusions and discusses future work.

2. Webor - A Search Tool Developed for this Research

The tool we developed and used in this research, Webor (Web-based search tool for Organization Retrieval) [Webor96] is based on the vector space model and uses the following Cosine formula for calculating the similarity between a query and a document [Salt83],

$$\text{Sim}(q, d)=\frac{\sum_{i=1}^{k} q_i w_i}{\sqrt{\sum_{i=1}^{k} q_i^2 \sum_{i=1}^{k} w_i^2}} \qquad (1)$$

where k is the dimension of the vector space, w_i is the weight of the ith term in the document vector d

and q_i is the weight of the *i*th term in the query vector *q*.

Webor contains an *indexing engine* and a *search engine*. Before we can describe how Webor works we must describe how and why some HTML tags were grouped into classes.

2.1 The Classes

We have grouped HTML tags into the following six classes: Plain Text, Title, H1-H2, H3-H6, Strong, and Anchor. The terms in the Plain Text class are terms that do not appear in the text enclosed by the title, header, or emphasized structures of an HTML document (see Table 1).

Table 1: The Six Classes and associated HTML tags

Class Name	HTML tags
Anchor	A
H1-H2	H1, H2
H3-H6	H3, H4, H5, H6
Strong	STRONG, B, EM, I, U, DL, OL, UL
Title	TITLE
Plain Text	None of the above

The reason that we grouped HTML tags into six classes, instead of having a separate class for each type of tag, was to reduce the size of the index, the work needed to find an optimal importance factor combination, and to improve the efficiency of Webor. We will now explain how the six classes were selected. At this point in the research there is no guarantee that this grouping is optimal.

We have divided the HTML header tags into three classes: Title, H1-H2 and H3-H6. The tags in the Title, H1-H2 and H3-H6 classes are TITLE, H1 and H2, and H3, H4, H5, and H6, respectively. The reasoning behind using these three classes was as follows: The terms in a document's title provide information on what a document is about, and thus should belong to a single class. The terms in the H1 and H2 headers provide descriptions of the main structure and topics of a document and thus should be grouped into a second class. The terms in the H3, H4, H5, and H6 headers provide information about the more specific structure and topics of a document and hence should be grouped into a third class.

We have grouped the HTML list tags, and the strong, emphasized, bold, underscored, and italic tags into a single class called the Strong class. The idea here was that terms that are emphasized and terms which appear inside lists, are terms that the author perceived to be important to the contents of the document and thus should be grouped together.

The Anchor class includes all the terms, which occur in the anchor tag of hyperlinks to the document. The justification for including this class in the index is that this information provides additional knowledge about the main subject of the document and should be taken into consideration when a query and a document are matched.

2.2 The Indexing Engine

The index built by Webor consists of a web page index and a keyword index. The web page index contains information concerning the web pages, including their IDs and URLs. The keyword index is organized into an inverted file for efficient retrieval. For each collection term *t* Webor keeps the total number of web pages that have *t* (*df*) and an inverted list. The inverted list for *t* is a sequence of pairs. The first element in each pair is the ID of some web page *d*, and the second element is a *Term Frequency Vector*, *TFV*. *TFV* contains the frequency of occurrence of *t* in each class associated with *d*. The term frequency vector is $TFV = (tfv_1, tfv_2, tfv_3, tfv_4, tfv_5, tfv_6)$ where tfv_1, tfv_2, tfv_3, tfv_4, tfv_5, and tfv_6 are the term frequencies of *t* in the Plain Text, Strong, H3-H6, H1-H2, Anchor, and Title classes, respectively.

Webor parses each word *s* of a document *d*, checks that *s* is not a stop word, and stems it. The result of the stemming process on a non stopword *s* is the term *t*. When Webor encounters the term *t* for the first time in document *d*, it generates a term frequency vector *TFV* = (0, 0, 0, 0, 0, 0) for *t*, and then determines a class assignment for *t*. To determine the class assignment, Webor uses the following precedence order: TITLE tag > H1&H2 tags > H3&H4&H5&H6 tags > Strong tags > None of the above (see Table 1). This means that when a word is enclosed in the TITLE tag it is assigned to the Title class regardless of any other tag in which it is enclosed, and only terms which are not enclosed by any of the header or Strong tags are assigned to the Plain Text class. Next, based on the assigned class, Webor increments one of the counts: tfv_1, tfv_2, tfv_3, tfv_4 , and tfv_6.

Indexing the terms of the Anchor class is the last process performed by Webor because it needs to collect all the anchor text in the collection with hyperlinks to document *d*. Webor uses another file to keep this information about the anchor text associated with *d*. After Webor visited and indexed all web pages in the collection, it then indexes this file for the Anchor class. Each occurrence of term *t* in anchor descriptions of hyperlinks to *d* is used to increment tfv_5.

2.3 The Search Engine

The *search engine* is a CGI (Common Gateway Interface) program, which takes a query from a Web user via an HTML form and returns to the user a ranked list of HTML hyperlinks. The query can be an AND, or an OR Boolean query, or a list of terms. The user can also input a weight for each term. In addition, users can limit the number of web pages returned to them.

To enable conducting retrieval experiments, Webor requires the user to provide the six class importance values, which will be used in the current experiment. These values are stored by Webor in the *Class Importance Vector* $CIV = (civ_1, civ_2, civ_3, civ_4, civ_5, civ_6)$, where civ_i is the importance factor assigned to class *i* in the current experiment.

The *search engine* parses the query and uses the inverted file index built by the *indexing engine*, and the *CIV* to retrieve all web pages whose *TFV* values are not all 0 for at least one of the query terms.

To take advantage of the *TFV* information produced by the *indexing engine*, and to take into account the *CIV* provided for the experiment, we needed to modify the computation of the weight *w* of term *t* in document *d* which Webor uses in the computation of the Cosine similarity (see formula (1)). Webor uses the formula $w = (TFV \circ CIV) \cdot idf$ where the inner product of the two vectors, *TFV* and *CIV*, represents the importance of term *t* to document *d*, and *idf* is the inverse document frequency of the term in the collection. For calculating *idf*, Webor uses the commonly used formula *idf* = *ln(N/df)* [Salt83], where *N* is the number of documents in the collection, and *df* (*document frequency*) is the number of documents that contain the term.

Finally the *search engine* sorts the retrieved documents by nonincreasing similarity and produces a ranked list of hyperlinks to WWW pages which the user can access.

2.4 Normal Retrieval and Normal CIV

Note that when the $CIV = (1,1,1,1,0,1)$, the weight *w* of term *t* reduces to:

$$w = (TFV \circ (1,1,1,1,0,1)) \cdot idf$$

$$= (tfv_1 + tfv_2 + tfv_3 + tfv_4 + tfv_6) \cdot idf$$

$$= tf \cdot idf$$

So with this *CIV* the weight calculated by Webor for each term is $w = tf \cdot idf$. In this case, the retrieval results obtained by Webor are equal to those obtained by any vector space based IR system that ignores HTML tags, uses $tf \cdot idf$ to calculate term weights, and Cosine to calculate the similarity between a query and a document. We call this *CIV* the *Normal CIV* and the retrieval with Normal *CIV*, the *Normal Retrieval*. The Normal retrieval results are compared to the results of experiments with other *CIVs*, and used to show the percentage of improvement achieved by better *CIV*s.

3. The Testbed

3.1 The Document Collection and the Queries

The document collection of the testbed includes all WWW pages that belonged to Binghamton University at the end of 1996. Webor's indexing robot was run with the domain seed "binghamton.edu" and indexed 4,596 HTML documents. The average number of words in an HTML document was 309. Table 2 shows the total number of term occurrences that were assigned to each class, and the percentage of these term occurrences. Note that the classes with the highest percentages are Plain Text (79.8%), Strong (13.2%), and Anchor (2.8%).

Table 2: The Distribution of term occurrences among the six classes

Class	Terms	Percentage
Anchor Class	39,840	2.8 %
H1-H2 Class	21,232	1.5 %
H3-H6 Class	27,266	1.9 %
Plain Text Class	1,131,376	79.8 %
Strong Class	187,094	13.2 %
Title Class	10,955	0.8 %
Total	1,417,763	100 %

The 10 queries used for the experiment (see the left column of Table 3) are the typical short queries used by faculty and students to find information in a university environment. Some of the queries relate to administrative issues, such as “promotion guidelines”, while other queries relate to subject matters such as “neural networks”.

3.2 Relevant Document Identification

To find the relevant set of documents for a given query, we substituted the query with a set of other queries (see column 2 of Table 3), and used Webor to create an expanded set of retrieved documents. This expanded set was checked manually to determine the subset of relevant documents. New queries were generated by using OR with synonyms, adding AND queries, and by omitting less important terms from the original query. For example, the query, “handicapped student help”, was expanded into the queries, “handicap OR disable” and “physical AND challenge”. These two queries enable Webor to retrieve all documents which refer to anyone with a disability. Note that the words “help” and “students” were omitted. This method enabled our finding the subset of relevant documents in the document collection for each of the ten queries.

We determined that a web page is relevant to a query if the web page was about the topic of the query, or was a resource (for example, a list of hypertext links) for finding information on the topic of the query. Column 3 of Table 3, shows the number of documents relevant to each query.

4. The Experiments

A large number of experiments were conducted to find an optimal *CIV* and compute the improvement it provides. The evaluation is based on the recall-precision metric widely used in information retrieval. For a given query, when a set of documents is returned from the IR system, the *recall* is defined to be the ratio (number of relevant documents retrieved)/(number of relevant documents in the collection) and the *precision* is defined to be the ratio (number of relevant documents retrieved)/(number of retrieved documents). The best retrieval effectiveness is achieved when both recall and precision are equal to 1. However, in practice, this is unlikely to occur. Usually, when higher recall is achieved, the precision becomes lower.

For each experiment we conducted, the ten testbed queries were used to retrieve documents from the testbed document collection based on a given *CIV*. For each query the retrieval results were evaluated at 11 recall points, starting at 0, ending at 1, and using increments of 0.1 Then, the precision results of the ten testbed queries were averaged at the 11 recall points. Finally the 11 precision values are averaged into a single number which we call the 11-point average precision. Traditionally, this number is used to represent the effectiveness of an information retrieval system. In this study, we have added the 5-point average precision, computed by averaging the testbed queries’ average precision at recall values of 0, 0.1, 0.2, 0.3, and 0.4 to provide additional important information about the effectiveness of the system. Because of the large and increasing number of web documents, web search tools often return a

Table 3: The Queries

Original Queries	Modified Queries	# of Relevant Docs
web-based retrieval	1. web-based OR retrieval 2. web AND search	15
neural network	neural AND network	26
master thesis in geology	Geology	3
prerequisite of algorithm	Algorithm	4
handicap student help	1. handicap OR disable 2. physical AND challenge	14
promotion guideline	Promotion	4
grievance committee	Grievance	17
laboratory in electrical engineering	1. electrical 2. laboratory	8
anthropology chairman	anthropology OR chairman	3
computer workshop	workshop OR seminar	16

very large number of retrieved documents. Unfortunately only a relatively small percentage of these documents are useful to the web searcher. To locate some good documents a searcher has to download and browse, or read a summary of a large number of these documents. Since the results are ranked the searcher usually starts with the first document retrieved by the system and then proceeds down the list until a sufficient number of good documents have been located. This is a very time consuming and frustrating process. Often web searchers do not need all good documents for a given query and are satisfied with finding a small number of good documents. This makes high precision for the top documents retrieved by the system very important. In other words, high precision at a lower level of recall is very useful in the WWW environment.

4.1 Retrieval with a Single Class

Our first set of experiments were to compare *Normal Retrieval* with retrieval based only on terms appearing in a single class using an importance factor of 1 (SC/1). As can be expected the results are inferior to those of *Normal Retrieval* (see Table 4). It is interesting to observe, however, what happens when just the Anchor class is used for retrieval. The 11-point average precision is only 5% below the *Normal Retrieval*, and is 33% better than the *Normal Retrieval* for the 5-point average precision. These results indicate the usefulness of the descriptions included in the Anchor class. It is also interesting to observe that the 5-point average precision is comparable to that of the 5-point *Normal Retrieval*, for the Strong class only and the Title class only experiments.

4.2 Finding a Good Importance Factor for one Class of a Normal CIV

To determine the effect of increasing the importance factor of a single class of a Normal *CIV*, we conducted experiments where a number of different importance factors were assigned to one class, while the rest of the factors remained as those of a Normal *CIV*. The importance factor of the plain text class was fixed at 1, and the experiments attempted to find a good (or optimal) value for each of the other classes. In this paper we only include the results obtained by 3 experiments conducted for the Anchor class (the results for the other classes are in [Shih97]).

The results of the experiments for assigning importance factor values 2, 4, and 6 to the Anchor class are shown in Table 5. Note that when the factor is either 2 or 6 the results are almost identical to those of the *Normal Retrieval*, but a factor of 4 gives a better average precision for both 5-point and 11-point average precisions. Compared to *Normal Retrieval* the improvement is 9% for 5-point average and 5% for 11-point average.

Table 6 shows the best results obtained by experiments that vary a single factor in the Normal *CIV*, for the Strong, H1-H2, Anchor and Title classes. It shows that by using a factor of 8 for the Strong class we get improvements of 10% and 7% in retrieval results over *Normal Retrieval*. The table does not include the H3-H6 class since in our experiments a Normal *CIV* with factors larger than 1 for the class H3-H6 does not provide better results.

Table 4: Improvement over Normal using SC/1 for retrieval

Class	*CIV*	5-Point Average Precision	11-Point Average Precision	5-Point Improvement over Normal	11-Point Improvement over Normal
Normal	111101	0.249	0.201		
Anchor only	000010	0.332	0.191	33%	-5%
H1-H2 only	000100	0.274	0.159	10%	-21%
H3-H6 only	001000	0.097	0.047	-61%	-76%
Plain Text only	100000	0.203	0.112	-19%	-44%
Strong only	010000	0.255	0.156	2%	-22%
Title only	000001	0.258	0.187	4%	-7%

Table 5: Improvement over Normal using Anchor factors 2, 4, and 6.

Class/Factor	*CIV*	5-Point Average Precision	11-Point Average Precision	5-Point Improvement Over Normal	11-Point Improvement over Normal
Normal	111101	0.249	0.201		
Anchor/2	111121	0.245	0.200	-1%	0%
Anchor/4	111141	0.271	0.211	9%	5%
Anchor/6	111161	0.245	0.199	-2%	-1%

Table 6: The Best Value of a Single Factor

Class	*CIV*	5-Point Average Precision	11-Point Average Precision	5-Point Improvement Over Normal	11-Point Improvement over Normal
Normal	111101	0.249	0.201		
Strong/8	181101	0.273	0.215	10%	7%
H1-H2/4	111401	0.274	0.213	10%	6%
Anchor/4	111141	0.271	0.211	9%	5%
Title/4	111104	0.272	0.213	9%	6%

4.3 Finding an Optimal CIV

Once a best factor value was determined for each single class in an otherwise Normal *CIV*, we tried the combination of all best pairs (with values greater than 1, i.e., the best pairs from the Strong, H1-H2, Anchor and Title classes) in an otherwise Normal *CIV*. Table 7 shows the improvement over Normal *CIV* achieved by using pairs of best factors in an otherwise Normal *CIV*.

Note that by using *CIV*s with best factor pairs for Strong&H1-H2, Strong&Anchor, and Strong&Title, the improvements are comparable to the sum of the improvements achieved with the best factors for the two corresponding individual classes. For example, when the single best factor for Strong is used the improvements are 10% and 7%, when the best factor for H1-H2 is used the improvements are 10% and 6%, and when both best factors for Strong and H1-H2 are used the improvements are 21%~10%+10% and 13%=7%+6%. The results indicate also that the Strong class is very important. Note that the improvement for the single best factor in the classes H1-H2, and Title are canceled when using the best pair for H1-H2&Title.

Other experiments conducted by us resulted in a better *CIV* = (181181) for the class pair Strong&Anchor in which the value of the Anchor factor was increased to 8. Additional experiments in which a single factor was changed in this *CIV* have not shown any further improvement.

We next experimented with changing the factors of the H1-H2 and Title classes of the *CIV* = (181181). The results are summarized in Table 8.

The best vector found was (181684), which improved the average precision over normal by 26% for the 11-point average precision, and by 44% for the 5-point average precision. Experiments with the effect of changing a single value in the *CIV* (181684) showed no improvements in the results. This was the best *CIV* that we have found.

4.4 Determining the Importance of Each Class

Another way to determine the importance of the terms in a given class, say C, is as follows. We first fix the importance factor for C to that as in the Normal *CIV*. Then we try to find the new best *CIV by* adjusting the importance factors for the other three classes (the importance factors for the plain text class and H3-H6 class are fixed at 1). If the retrieval effectiveness based on the new best *CIV* is about the same as that based on the old best *CIV* (i.e., (181684)), then this indicates that the terms in C are not very important for enhancing retrieval effectiveness. On the other hand, if the retrieval effectiveness based on the new best *CIV* is substantially lower than that based on the old best *CIV*, then the terms in C are very important for improving retrieval effectiveness as adjusting the importance factors for the other classes alone can not achieve the same level of improvement. The results of this set of experiments are summarized in Table 9. From this table, it is clear that the Strong class and the Anchor class are very important while the H1-H2 class and Title class are less important.

5. Conclusions and Future Work

In this paper, we proposed a method for making use of the structures and hyperlinks of HTML documents to improve the effectiveness of retrieving HTML documents. Our method partitions the occurrences of terms into six classes (title, H1-H2, H3-H6, anchor, strong and plain text) and adjusts the traditional term weighting scheme by incorporating different importance factors to term occurrences in different classes. Through extensive experiments, we showed that by using our method it is possible to substantially improve the retrieval effectiveness (see Table 8). In particular, we found that the terms in the Strong and Anchor classes are the most useful for improving the retrieval effectiveness.

Table 7: Improvement by using *CIV*s with Best Factor Pairs

	CIV	**5-Point Average Precision**	**11-Point Average Precision**	**5-Point Improvement Over Normal**	**11-Point Improvement over Normal**
Normal	111101	0.249	0.201		
Strong&H1-H2	181401	0.300	0.228	21%	13%
Strong&Anchor	181141	0.300	0.226	21%	13%
Strong&Title	181104	0.296	0.226	19%	13%
H1-H2&Anchor	111441	0.274	0.213	10%	6%
H1-H2&Title	111404	0.248	0.202	0%	0%
Anchor&Title	111144	0.268	0.210	8%	4%

Table 8: Improvement over Normal with *CIV*s (181181) and (181684)

CIV	**5-Point Average Precision**	**11-Point Average Precision**	**5-Point Improvement over Normal**	**11-Point Improvement over Normal**
Normal	0.249	0.201		
181181	0.353	0.251	42%	25%
181684	0.357	0.254	44%	26%

Table 9: The Improvement by each factor of the best *CIV*

Class with Factor fixed to Normal	***CIV***	**5-Point Improvement Over Normal**	**11-Point Improvement Over Normal**	**Contribution of factor in best *CIV* for 5-point**	**Contribution of factor in best *CIV* for 11-point**
Best *CIV*	181684	44%	26%		
Strong to 1	111644	11%	7%	33%	19%
H1-H2 to 1	181181	42%	25%	2%	1%
Anchor to 0	181604	21%	14%	23%	12%
Title to 1	181681	42%	25%	2%	1%

We plan to conduct more experiments using an expanded set of queries and possibly different web page collections. We believe that substantially more extensive experimental results need to be collected and analyzed in order to assess accurately the effectiveness of using HTML structures. Another issue to investigate is the optimal assignment of tagged information to classes. It is possible that information in lists should be in a different class from emphasized terms, or that all headers H1-H6 should be included in one class. In this study, the similarity function used is Cosine, and the term weight function is a modification of $tf \cdot idf$. Other similarity and term weight functions have also been used in traditional IR systems. We are interested in examining how different similarity and weight functions may affect the retrieval effectiveness. Studying the feedback process in the WWW environment is also of interest.

6. Availability

The information about Webor can be accessed at

http://nexus.data.binghamton.edu/~yungming/webordoc.html

It also includes links to download the testbed.

References

[Agost96] M. Agosti and A. Smeaton, *"Information Retrieval and Hypertext"* Kluwer Academic Publishers, 1996.

[Alta96] Digital Equipment Corporation, "*ALTA VISTA: Main Page*", http://altavista.digital.com/cgi-bin/query/, 1996.

[CMU95] Carnegie Mellon University, "*Lycos, The Catalog of the Internet*", http://lycos.cs.cmu.edu/, 1995.

[Croft93] W.B. Croft and H.R. Turtle, *"Retrieval strategies for hypertext"* Information Management and Processing 29(3), 1993, 313-324.

[Dunlop93] M. D. Dunlop and C. J. Van Rijsbergen "Hypermedia and free text retrieval", Information processing and Management 29(3), 1993, 287-292.

[Frei92] H.P. Frei, D. Stieger , *"Making Use of Hypertext Links when Retrieving Information"*, Proceedings ACM ECHT'92, Milan, Italy, 1992, 102-111.

[Frisse88] M.E. Frisse, *"Searching for Information in a Hypertext Medical Handbook"* Communications of ACM 31(7) July 1988.

[HotBot] Inktomi, Inc., HotBot Home Page, http://www.hotbot.com/.

[Hypur96] R. Weiss, B. Velez, M.A. Sheldon, C. Nemprempre, P. Szilagyi, A. Duda, and D.K. Gifford, *"HyPursuit: A Hierarchical Network Search Engine that Exploits Content-Link Hypertext Clustering"*, Proceedings of the Seventh ACM Conference on Hypertext, Washington, DC, March 1996.

[Mage97] The McKinley Group Inc., *"Magellan Internet Guide"*, http://www.mckinley.com/, 1997.

[Mau97] M.L. Mauldin, *"Lycos: Design choices in an Internet search service"*, IEEE Expert Online, February 1997.

[Mi97] Microsoft Co., *"microsoft.com Search Wizard"*, http://www.microsoft.com/search/default.asp, 1997.

[OTC96] OpenText Corporation, "*Search the World Wild Web – Every word every page*", http://www.opentext.com:8080/, 1996.

[Pi94] B. Pinkerton, "*Finding What People Want: Experiences with the WebCrawler*", Proceedings of the 2nd Int'l World Wide Web Conf., Elsevier Science, http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/WWW2_Proceedings.html, 1994.

[Salt83] G. Salton and M.J. McGill, *"Introduction to Modern Information Retrieval"*, McGraw-Hill, New York, NY, 1983.

[Salt89]. G. Salton, *"Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information* *by Computer",* Addison Wesley, 1989.

[Shih97] Y. Shih, "A Study of the Usefulness of the Structure of HTML Documents on Retrieval Effectiveness", MS thesis, Dept. of Computer Science, State of New York at Binghamton, 1997.

[Webor96] J. Lu, Y. Shih, W. Meng, and. M. Cutler, *"Web-based search tool for Organization Retrieval*", http://nexus.data.binghamton.edu/~yungming/webor.html, 1996.

[Yaho96] Yahoo Inc., "*Yahoo Search*", http://www.yahoo.com/search.html, 1996.

[YuLe96a] B. Yuwono and D.L. Lee, *"Search and Ranking Algorithms for Locating Resources on the World Wide Web"*, Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, Feb. 26 - March 1, 1996, 164-171.

[YuLe96b] B. Yuwono and D.L. Lee, *"WISE: A World Wide Web Resource Database System"* IEEE Transactions on Knowledge and Data Engineering, Special Section on Digital Library, Vol. 8, No. 4, Aug 1996, 548-554.

SASE: Implementation of a Compressed Text Search Engine

Srinidhi Varadarajan Tzi-cker Chiueh

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400

(srinidhi, chiueh)@cs.sunysb.edu

http://www.ecsl.sunysb.edu/RFCSearch.html

Abstract

Keyword based search engines are the basic building block of text retrieval systems. Higher level systems like *content sensitive* search engines and knowledge-based systems still rely on keyword search as the underlying text retrieval mechanism. With the explosive growth in content, Internet and Intranet information repositories require efficient mechanisms to store as well as index data. In this paper we discuss the implementation of the **Shrink and Search Engine** (*SASE*) framework which unites text compression and indexing to maximize keyword search performance while reducing storage cost. *SASE* features the novel capability of being able to directly search through compressed text without explicit decompression. The implementation includes a search server architecture, which can be accessed from a Java front-end to perform keyword search on the Internet.

The performance results show that the compression efficiency of *SASE* is within 7-17% of GZIP one of the best lossless compression schemes. The sum of the compressed file size and the inverted indices is only between 55-76% of the original database while the search performance is comparable to a fully inverted index. The framework allows a flexible trade-off between search performance and storage requirements for the search indices.

1. Introduction

Efficient search engines are the basic building block of information retrieval. Content sensitive engines like Lycos and Yahoo still rely on keyword search as their underlying search mechanism. Furthermore, with growth in corporate intranet information repositories, efficient mechanisms are needed for information storage and retrieval.

In this paper we propose a scheme to maximize keyword search performance while reducing storage cost. The basic idea behind the proposed framework called the **Shrink and Search Engine** (*SASE*), is to use the commonality between dictionary coding and inverted indexing to unite compression and text retrieval into a common framework. The result is a search engine that is efficient both in terms of raw speed as well as storage requirement, and has the capability of searching directly through compressed text.

This paper is organized as follows. Section 2 describes the basic idea behind *SASE*. In section 3 we discuss the implementation issues and our Internet *SASE* Server architecture. Section 4 reports the results of a performance analysis of our system. In section 0, we present related work in the area. Section 6 concludes the paper with a report on the major results and future work in the area

2. Basic Algorithm

The common approach to fast indexing uses a structure called the *inverted index*. An inverted index records the location of each word in the database. When a user enters a query word, the inverted index is consulted to get occurrence list of the word. Typically the inverted index is maintained as a dictionary with a linked list of occurrence pointers associated with each word. The dictionary is organized as a hash table for faster keyword search.

A significant characteristic of textual data is the high degree of inherent redundancy in it. Text compression reduces source redundancy by substituting repetitive patterns with shorter numerical identifiers. Text compression can be done by variable bit length statistical schemes like Huffmann coding or dictionary based schemes like LZW, which substitute identical character strings with dictionary identifiers representing the pattern. Our observation here is, that both inverted indexing and dictionary based text compression require a dictionary. Hence one can reuse the dictionary from the inverted index for dictionary coding uniting compression and pattern matching into a common framework.

Dictionary based compression can be done at several levels of token granularity. In our united compression/pattern matching framework, we use a *word* as the basic dictionary element. A word is any pattern punctuated by white-space characters. The advantage of this approach is that it integrates the requirements of word based pattern matching and compression. The drawback is that the compression efficiency is not as high as that obtained from dictionary schemes like Lempel-Ziv which use arbitrary string tokens.

Text compression is performed in *SASE* by substituting words with their numerical representation called *lexical codes*. To improve the utilization efficiency of the available lexical code space, we use a technique similar to Huffmann coding at the byte level. The set of words in a database is partitioned into three groups viz. *common words*, *uncommon words* and *literals*. Common words occur more frequently than uncommon words, which in turn occur more frequently than literals. The classification is done on the basis of the *compression benefit factor* (CBF) of a word, which is defined as the product of the length of the word and its occurrence count. This partitioning is done off-line since the target applications for this scheme are mainly read-only databases. In the common word dictionary, words are represented by a 1 byte code. The uncommon word and literal dictionaries use a 2 byte code. Our experiments show that common words occur more than 50% of the time and greatly benefit from their smaller representation.

2.1 Compression and Decompression

In order to compress a text database, the database is first scanned to determine the list of unique words sorted by their compression benefit factors. The first 256 words are put in the common word dictionary and the next 64K words are put in the uncommon word dictionary. The second pass is done during the compression phase where each word in the database is converted to its dictionary id. In this pass literals are identified and literal dictionaries are created on demand. This scheme allows us to share the common and uncommon word lists across multiple *similar* databases. Compression on such databases would need only one pass.

The compressed representation of a text file consists of the following four files:

1. **.cw* : A file of common-word dictionary IDs, each of which is represented as a 1-byte codeword indexing into the common word dictionary. There are some exceptions. Ten of the 256 1-byte codewords are used as special flags to indicate that the next word is a literal whose 2 byte code is in the literal file. Some other codes are used to optimize capitalization and for run-length-encoded tokens, as explained in Section 3.1
2. **.ucw*: A file of uncommon-word dictionary IDs, each of which is represented as a 2-byte codeword indexing to the uncommon word dictionary.
3. **.lit*: A file of literals, each of which is represented by a 2-byte codeword indexing to the literal dictionary.
4. **.bit*: A bitmap file in which each bit represents a word in the text database and indicates whether it is a common word/literal or an uncommon word.

Fig. 1 shows the compressed representation of the string "*There was an ugly aardvark in the room*". The words *there, was, an, in* and *the* are assumed to be common words and are assigned the dictionary ids 1, 2, 3, 4 and 5 in the common word dictionary. Similarly *ugly* and *room* are uncommon words and are assigned the ids 1 and 2 in the uncommon word dictionary, whereas the word *aardvark* is a literal and is assigned the code 1 in the literal dictionary 1. In the compressed representation of the string, the bitmap file is used to direct the decompression engine to go to either the compressed common word file or the uncommon word file. To get the next code from the literal file we indicate that the next word is a common word and then use a special code in the common word file to further direct the decompression engine to get the next word from the literal file. Codes 239 to 249 are reserved in the common word file to direct decompression to literal dictionaries 1 to 10.

While this scheme is roughly modeled on the lines of Huffmann coding, it has two distinct advantages over Huffmann coding. First the code space is used more efficiently since individual dictionary ids do not have

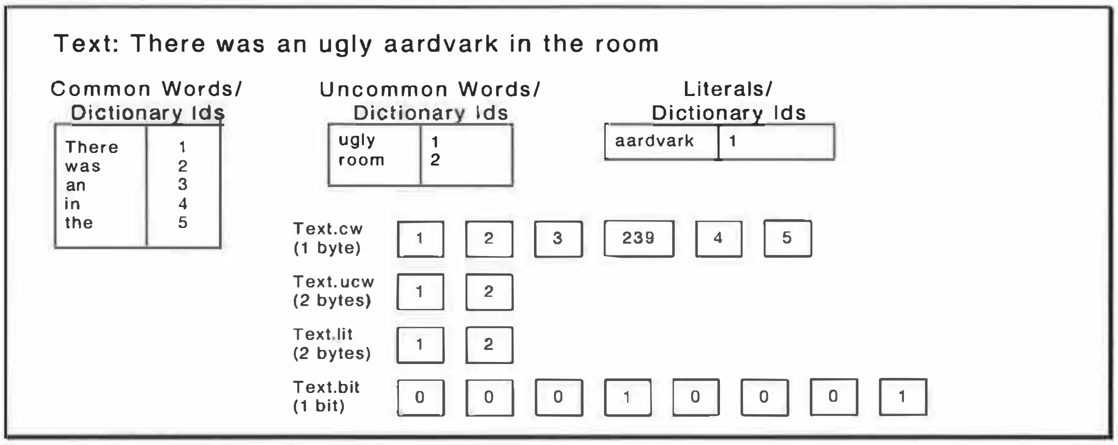


Figure 1: The compressed text representation of an example string. The literal word "aardvark" is represented by a reserved code 239 in the common word file

to satisfy the unique prefix property. Secondly codes of different length for different dictionaries are maintained in independent files. A bitmap file consisting of 0s and 1's is used to direct the decompression scheme. A 0 indicates that the next word is in the common word file whereas a 1 indicates that the next word is in the uncommon word file. A reserved code in the common word file may further indirect the decompression to read the next code from the literal files. This scheme can be considered an instance of Huffmann coding with a 1 bit prefix.

The number of unique words found in *normal* databases (stories, newspaper articles etc.) is quite small. However, technical databases tend to have a very large vocabulary, particularly when they contain computer program code or ASCII art. To accommodate these words, *SASE* reserves code space in the common word file to support up to 10 literal word dictionaries of 64K words each for a total of 704K words. More codes may be reserved to support larger databases at a small penalty in compression ratio due to the increased number of reserved tokens.

2.2 Indexing and Searching

In a full inverted index structure, each dictionary entry consists of a linked list, which records the positions of all instances of the word. When the user enters a keyword, the linked list is followed to obtain all the occurrences of a keyword. While this scheme has very fast search times, the space complexity of generic full inverted indexing schemes is quite large. The size of the inverted index has been reported to range between 50%-300% of the size of the original database[FALO85].

SASE solves this problem by using an indexed approach. The text database is partitioned into blocks by partitioning the bitmap file into equal sized chunks. The pointers in the linked list are block identifiers. Note the partitioning is in terms of bits in the bitmap file, for example a 4KB block size contains 4K*8 = 32K words. The first occurrence of a keyword in each block is recorded irrespective of the number of occurrences of the keyword in the block. In order to reach the other occurrences, a linear search is performed on the block. This scheme allows a flexible trade-off between speed and storage requirement. With a smaller block size, it takes less time to search through it, whereas the space requirement increases since there are more block pointers. Conversely, a larger block size requires less block pointers whereas the time required for searching is larger.

In the indexed approach taken by *SASE*, we need to perform a linear search in a block to find other instances of a keyword. A naïve implementation would decompress the compressed text and perform string comparisons between the query word and the decompressed text. Since *SASE* applies dictionary coding in its compression scheme, it is possible to **search directly through the compressed text** without explicit decompression. The query keyword is first converted into its dictionary id and directly compared against the dictionary id's in the compressed text. When an instance of the keyword is found, the location in the compressed files is marked. Future searches can begin from this location. Search within a block is terminated when the number of instances of the keyword found matches the count field associated with a block, which maintains the total number of occurrences of the keyword. This scheme is much faster than any string comparison based indexing scheme since we only need to perform fixed length numeric comparisons as opposed to variable length string comparisons.

Boolean queries can be performed by AND/OR operations on the linked lists associated with the query keywords. The resultant list formed by the applying the Boolean expression on the linked lists is then searched.

The block size of the inverted index plays a critical role in the performance of *SASE.* An optimization that can be performed here is to use different block sizes for different words. *SASE* implements a fully indexed **dynamic index cache** to reuse results from previous searches. A separate dictionary is used which caches every occurrence of the most frequently/most recently accessed words. When a keyword is searched, the search results are posted to the index cache. Since *SASE* supports next occurrence type of queries, it is possible to have incompletely filled entries in the index cache. These incompletely filled entries are filled when a user query accesses all occurrences of a keyword. The index cache is consulted to see if it can satisfy a request before beginning a search using the inverted index.

2.3 Approximate Search

For approximate searching, the set of uncommon words and literals are statically organized in a **Vantage Point** (VP) [YIAN92][CHIU94] tree. The user specifies the desired maximum number of errors between the query word and his results. We can then traverse the VP tree to get a set of words that fall within the allowable error range.

The set of allowable branches is determined by comparing the query word against the interior nodes of the tree. The remaining branches are pruned since we know that none of their leaf nodes can contribute to the query. Although this scheme performs considerably better than a linear search through the dictionary, the number of comparisons is still high. An interesting observation here is that word lengths are finite and discrete. Hence, we can build multiple VP trees, one for each length. When the user enters a query, the set of allowable VP trees is determined from the length of the query word and the desired maximum number of errors. These VP trees are then searched to get the set of allowable words. Experiments on this scheme show that we need to compare against 4-8% of the words in the dictionary to get the set of allowable words. After the allowable set of words has been determined, we search the database for each word in the set.

3. Implementation

The ITCI compression/search engine has been implemented in C running on a UNIX platform. It is consists of a (i) compression and decompression engine and (ii) a search engine. In our current implementation of *SASE,* we have built a communication subsystem around the search engine to allow searches from the World Wide Web using a Java front-end. In this section we discuss implementation details of the various sub-systems within *SASE.*

3.1 Compression Engine

Before we begin compression, we need to collect statistics to determine the word breakup into common words; uncommon words and literals based on the compression benefit factor. Once these statistics are collected, the compression engine builds up a hash table of common words and uncommon words. Literal hash tables are created *on demand* whenever they are encountered in the text. In this phase we also build an inverted index for uncommon words. Literal inverted indices are created on demand.

After the indices have been built, a parser parses the input token stream to extract words from it. Words are defined as a stream of alphanumeric characters delimited by white space tokens. Several optimizations are performed in this phase.

1. In a stream of natural text, a space character follows each word. It would be wasteful to store a token to represent the space. *SASE* assumes implicitly that each word is followed by a space. Absence of space is encoded with a reserved common word DELETE_SPACE flag, which precedes a token not followed by a space. This allows for better compression under the common case that each word is followed by a space

2. In a text database both normal and capitalized versions of a word can occur. Typically each sentence begins with a word whose first character is capitalized. Multiple versions of the same word take up additional dictionary space. For instance *version, Version* and *VERSION* would appear as three independent tokens. To prevent this, we precede capitalized words with a reserved common word code indicating the type

of capitalization; i.e. either the first letter or the entire word is capitalized.

3. In typical usage, a sentence is ended by a period after the last word in the sentence with no intermediate space between the period and the word. If we follow the above optimizations, at the end of a typical sentence we would end with a DELETE_SPACE code, followed by a code for the last word and a code for the period character. To optimize this case, we special case periods into normal periods and end of sentence periods. Tokens followed by an end of sentence period do not have a space between them by default. This saves us the byte required for the DELETE_SPACE code. This optimization is also used for commas and semicolons.

4. Many documents have repeating sequences of white space or punctuation characters for typesetting purposes or ASCII art. These tokens take up a lot of space in a dictionary since there are independent tokens for sequences of each length, for example the following sequences ------ and --------- used in ASCII tables represent two tokens in the inverted index. In most cases, these tokens are never searched. To optimize this, we perform Run length encoding (RLE) on space tokens and punctuation characters. Contrary to the 1 byte representation used in the common word file, a run length encoded token uses up 3 bytes in the common word file. The first is a reserved byte indicating that the next two bytes have to be treated as a run length encoding. The second byte contains the run length character and the third byte contains the length of the run. Since we use a 3 byte compressed representation, only runs greater than 3 bytes are run length encoded.

The compression engine gets tokens from the parser. If the token is not a reserved code, it searches the hash tables to determine if it is a common word, uncommon word or literal and gets the equivalent numerical representation. This numerical representation is then written to the appropriate file and the inverted index is updated to indicate this occurrence of the keyword. This proceeds till the input text database has been scanned and compressed.

An added feature of the compression engine is its ability to maintain document boundaries in a multi document database. This allows us to reconstruct the original documents from a multi-document compressed database.

3.2 Search Engine

The basic operation of the search engine is quite simple. When the user enters a keyword for searching we execute a hash based search function on it. If the keyword is found, the hash table returns the classification (common word, uncommon word or literal) and its numerical representation. Once we have the numerical code, we can index into the inverted index to get the linked list of pointers to occurrences of the keyword.

Since the inverted index is usually quite large, it is maintained on disk. When a keyword is found in the hash table, the corresponding inverted index entry is retrieved from disk. We have a block marker file associated with each compressed database, which marks the positions of the file pointers to the common word uncommon word and literal files at the start of each block.

To locate the first instance of a keyword, we go through the linked list to obtain the pointer to the block containing the keyword. Based on the information from the block marker file, we reposition the file pointers to the start of a block and perform a linear search to locate the keyword.

Typical queries ask for the first occurrence of a keyword, the next occurrence and so on based on the results obtained. A naive way to implement this would be to continue searching within the block till the occurrence number required by the user is found. For e.g. if user requests the third occurrence of *aardvark*, we search the block until we hit the third occurrence, ignoring the first two occurrences. While this solution works, it is hardly optimal.

In order to handle *next occurrence* kind of queries, we need maintain positional metadata on the previous location where the keyword was found. If we need to find the next occurrence of the keyword, we use this to reposition our *start of search* location. This solution allows us to perform incremental searching within a block with minimal time overhead.

After an instance of the keyword is found, we know the locations within the common word, uncommon word and literal files. We then backtrack on the common word, uncommon word and literal files till we can decompress a block of text (200 words in our case) around the occurrence and return it to the user. The backtracking algorithm is complicated by the fact that tokens in the common word file cannot be interpreted in the reverse order since there may be run

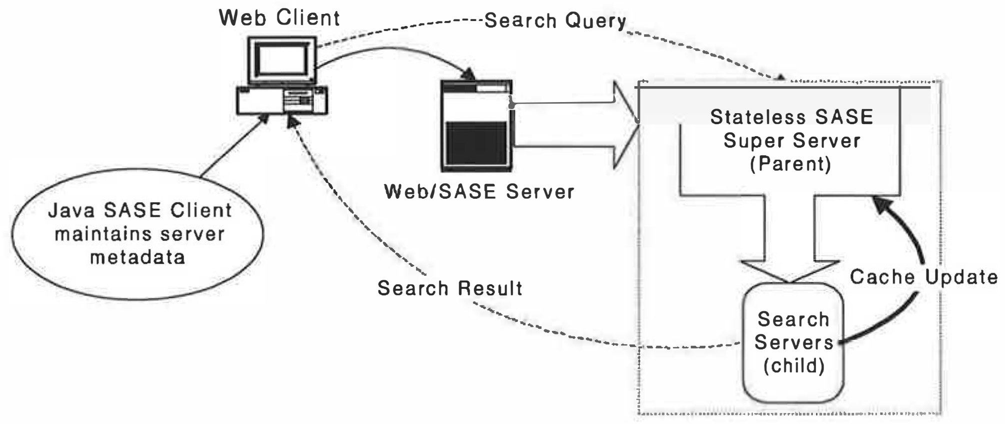


Figure 2 : Execution of a typical search query from then Web. The Java client sends in its query to the *SASE* super server, which forks a copy of itself to perform the search. The child process returns the result to the client along with server metadata. The child also performs a cache update on the super server.

length encoded tokens. Another complication is that we need to look out for document boundaries during the backtrack phase. In our scheme, backtracking is performed by using a lookback buffer for the common word file. This ensures that the numerical codes are interpreted correctly. The lookback buffer is not needed for the uncommon word and literal word files since these codes can be interpreted correctly in both directions.

3.3 Search Server Architecture

In our current implementation of the search engine, we have incorporated a communication sub-system, which allows queries to the search engine from the World Wide Web using a Java front-end. Communication is done using sockets opened between the Java client and the search server. For each query from a client, the server forks a copy of itself to perform the actual search and return the results. The server itself does a "busy wait" waiting for connections. The advantage of this scheme is that we use the semantics of the fork call to transfer the cache to the child process without performing a data copy. Most current UNIX systems implement the copy-on-write protocol. This reduces the overhead of the fork call. While this scheme allows a search child to see the same cache as the server, we need a mechanism to update the cache on the server when the child finds a new occurrence of a keyword. This is done by opening a named pipe between the server and its children. The children send back cache updates on the pipe and the server integrates it into its cache where it can be seen by future children. The update operation is sent as a block. Since the block is smaller than 4KB, UNIX named pipe semantics guarantee the atomicity of the update. This ensures that multiple simultaneous cache updates do not confuse the server. As mentioned earlier in this section, we need to maintain file position metadata after each keyword search to optimize *next occurrence* queries. . We use a novel scheme to maintain this metadata. In the *SASE* server architecture, the Java clients who send in this information with each next occurrence query maintain this metadata. The major implication of this scheme is that the server is now stateless, since each client knows its version of the server state.

In our Web communication model, the Java clients open a TCP socket to the server and send in their query and the server metadata. To prevent the clients from using up all the server connections, the socket is closed once the search engine returns its results. The small lifetime of sockets combined with the stateless nature of the server allows us to shut down the server and bring it up again without any noticeable difference to the clients. This is an attractive feature for administrative purposes. The downside of this scheme is that if the search database is updated, then the metadata on all the clients has to be updated as well. Since the clients close the socket once they get a response, we have no way of intimating them of changes in the server since we don't know who the clients are. This is solved by in a novel way. The compression engine generates a set of checksums

Database	Original Size	*SASE* Size	Gzip Size	Comp. Ratio (*SASE*)	Comp. Ratio (Gzip)
Stories	6,944,363	3,125,644	2,625,350	54.99%	62.19%
RFC	68,940,062	28,610,430	18,272,426	58.49%	73.48%
News	314,572,800	166,411,289	110,463,639	47.09%	64.88%

Table 1: Comparison of compression ratios under *SASE* and GZIP. All file sizes are in bytes.

Database	Original Size	*SASE* Size	Glimpse Size	Comp. Ratio (*SASE*)	Comp. Ratio (Glimpse)
Stories	6,944,363	3,125,644	5,340,213	54.99%	23.10%
RFC	68,940,062	28,610,430	48,365,620	58.49%	29.84%

Table 2: Comparison of compression ratios under *SASE* and *Glimpse*. All file sizes are in bytes.

Database	Total Time	Linear search (iii)	Repositioning (iv)	Decompression (v)
Stories	27.8 ms	26.1 ms	220 us	1.4 ms
RFC	39.1 ms	36.8 ms	280 us	1.8ms
News	41.2 ms	38.8 ms	240us	1.7ms

Table 3: Timing breakdown of the various steps involved in a search. Block size is 8KB

when it compresses a database. These checksums are sent to the clients when they first query the server and the clients then send in this checksum to the server with each message. If the database on the server changes in between queries from a client, the checksum at the server would change as well. On the next query from a client, the checksum maintained by the client would not match the checksum at the server. The server then sends a flush message to the client, forcing the client to flush its old metadata. With this scheme built into a stateless server, *SASE* can handle very large inter query times without any server overhead.

4. Performance Results

In this section we report the results of a performance evaluation of the *SASE* prototype and an analysis of the results. To evaluate our compression efficiency, we compare *SASE* against GZIP one of the best lossless compression utilities. We also present the search performance numbers of *SASE*.

To ensure a representative text database, we chose [1]three large text databases each representing of a certain vocabulary. The first database consists of 7 MB of stories from Project Gutenberg. This represents everyday literary English usage. The second database contains the Internet RFC documents. This 70MB database represents technical vocabulary from a particular field, in this case networking. The third database consists of 300MB of USENET news articles from several different newsgroups from different newsdomains like alt, rec, misc and comp. This database contains vocabulary from a wide variety of domains.

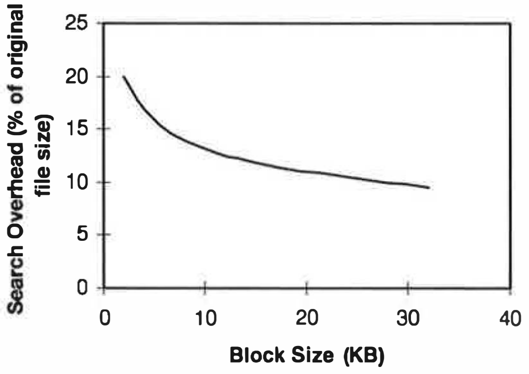
[1] Availability of large text databases poses a problem in itself

4.1 Compression Performance

Table 1 compares the compression ratios of *SASE* and GZIP. Compression ratio is defined as:

$$\text{Compression Ratio} = 1 - \frac{\text{Compressed File Size}}{\text{Original File Size}}$$

The differences range between 7 and 17%. Although both *SASE* and GZIP are based on dictionary coding, GZIP can choose arbitrary length strings as candidate tokens for compression. Since *SASE* is limited to choosing strings demarcated by white spaces as tokens it suffers a performance penalty in compression. This performance gain in GZIP is more marked in the News and RFC databases, which have a more dynamic vocabulary.



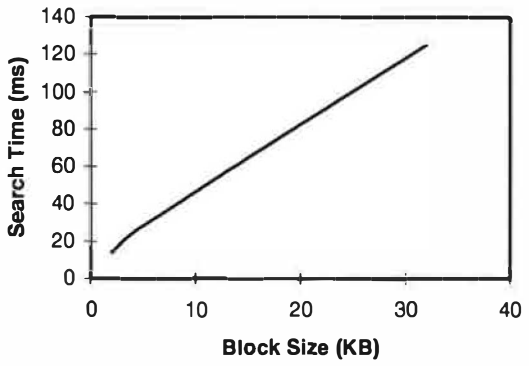


Figure 3: Effect of varying block size on the search overhead and query response time for the RFC database. Block size is varied betwen 2KB and 32KB.

4.2 Search performance

Execution of a search query under *SASE* involves (i) finding the numerical representation of the query keyword (ii) locating the first block containing the keyword (iii) performing a linear search to get the exact location of the keyword (iv) reposition of file pointers to begin decompression and (v) decompressing and transmitting the result back to the client. Steps (i) and (ii) are trivial and take much less time compared to the linear search required to locate the keyword within a block. Table 3 shows the timing breakdown of the various steps involved in a search.

As we mentioned in Section 2.2, by varying the block size, *SASE* allows a trade off between search time and index space overhead Figure 3 shows the effect of varying the block size on the RFC database. The index space overhead consists of the space taken up by the inverted indices for the uncommon word and literal dictionaries. The query search times were measured by searching for 10,000 random keywords and the runs were repeated for block sizes between 2KB and 32KB. Search times vary between 13ms to 120ms which compares favorably with a fully inverted index scheme.

5. Related Work

In [MOFF95][WITT94] [ZOBE95], Moffat and Zobel describe a word based lossless compression scheme which uses a fully inverted search index. The database is divided into files and compressed using Huffmann coding. Given a keyword, the inverted index is searched to get the linked list of occurrences. The portions of the file containing the keyword are then decompressed and sent to the user. Since this scheme uses a fully inverted index, the space taken up by the inverted index is much larger than *SASE.* Decompression speed is also slower than *SASE* due to the bit level manipulation that is required for decompressing Huffmann coded files. Variants of this scheme partition the database into blocks and compress the blocks using gzip. These schemes maintain inverted index pointers to blocks rather than every occurrence of a keyword. While these schemes have very good compression ratios, they need to decompress a block before searching through it. This operation increases their search time to several orders of magnitude greater than *SASE.*

The two step indexed approach taken by *SASE* is very similar to *Glimpse* [MANB94a][MANB94b]. Table 2 compares the compression efficiency of *SASE* and *Glimpse* (version 4.0) on both natural language text and technical documents. The difference in compression efficiency ranges between 28-31%. At the setting for the fastest search performance, the glimpse inverted index is 10% larger than SASE for comparable search times. *SASE* also optimizes the two level approach with an index cache of dynamic block size which allows to use a large block size for space savings while retaining the speed advantages of a smaller block size. On the other hand, glimpse does better job in the choice of keywords to index. The approximate pattern matching algorithm in *glimpse* (*agrep*) is also more powerful than the simple keyword search mechanism of *SASE.* Since our VP tree based, approximate keyword match framework takes the string comparison function as a black box; *agrep* could be used to search the inverted index to provide similar pattern patching capability in *SASE.*

There are also several theoretical studies [AMIR96][FARA95] which discuss algorithms for searching through Lev-Zempel files. However, these schemes have not been implemented for us to make a fair comparison.

6. Conclusions

In this paper, we described a text search engine called *SASE*, which operates in the compressed domain. It provides an exact search mechanism using an inverted index and an approximate search mechanism using a vantage point tree. Secondly it allows a flexible trade-off between search time and storage space required to maintain the search indices. The results of our experiments show that the compression efficiency is within 7-17% of GZIP, which is one of the best lossless compression utilities. The sum of the compressed file size and the inverted indices is only between 55-76% of the original database, while the search performance is comparable to a fully inverted index.

We are currently working on the implementation of the approximate search mechanism using a vantage point tree. Another area of work is to use *SASE* as the underlying file system for NNTP servers. This gives NNTP servers the capability to perform keyword searches through USENET archives. When this system is up, it would yield important results on the choice of a cache replacement policy for the *SASE* dynamic index cache. While incremental additions to the compressed database are permitted in an inverted index based search system, the database is assumed to be mainly read-only. We are working on an indexing mechanism that would effectively remove the read-only restriction and allow the user to make real time changes to the database without having to recalculate the inverted indices. This scheme can be used for document management servers within companies and for maintaining the web page index database in Internet search engines like Lycos and AltaVista.

References

[AMIR96] Amir, A., Benson, G., Farach, M.; "Let sleeping files lie: pattern matching in Z-compressed files", *Journal of Computer and System Sciences (April 1996) vol.52, no.2, p. 299-307.*

[BLUMER87] Blumer A., Blumer J.; "On-Line Construction of a Complete Inverted File", *Technical Report, Dept. of Mathematics and Computer Science., University of Denver, CO*

[BLUMER84] Blumer A., Blumer J., Ehrenfeuchter A., Haussler D., McConnell R.; "Building a Complete Inverted File for a Set of Text Files in Linear Time", *Proceedings of the Sixteenth Annual ACM Symposium on the Theory of Computing.*

[CHIU94] Chiueh T.; "Content-based image indexing" *Proceedings of VLDB '94 pp. 582-593, Santiago Chile, September 1994*

[EVEN78] Even S., Rodeh M.; "Economical Encoding of Commas Between Strings", *Communications of the ACM 21:4, 315-317*

[FARA95] Farach, M., Thorup, M.; "String matching in Lempel-Ziv compressed strings", *Proceedings of Symposium of Theory of Computing, pp. 703-712. Las Vegas, Nevada, USA.*

[FALO85] Faloutsos, C.; "Acccess methods for text", *ACM Computing Surveys, 17(March 1985), pp. 49-74.*

[FRAENKEL83] Fraenkel A.S., Mor M.; "Is Text Compression by Prefixes and Suffixes Practical", *The Computer Journal 26:4, pp. 336-344*

[KNUTH85] Knuth D.E.; "Dynamic Huffman Coding", *Journal of Algorithms 6, pp. 163-180.*

[KNUTH77] Knuth D.E. Morris J.H., Pratt V.R.; "Fast Pattern Matching in Strings", *SIAM Journal on Computing 6:2, pp. 323-349.*

[LEMPEL77] Jacob Ziv, Abraham Lempel; "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory, Vol. IT-23, No.3, May 1977.*

[MANB94a] Manber, U.; "A text compression scheme that allows fast searching directly in the compressed file", *Combinatorial Pattern Matching. 5th Annual Symposium, CPM 94. Proceedings, pp. 113-24. Asilomar, CA, USA, 5-8 June 1994.*

[MANB94b] Manber, U., Sun Wu; "GLIMPSE: a tool to search through entire file systems", *Proceedings of the Winter 1994 USENIX Conference, p. 23-32. San Francisco, CA, USA, 17-21 Jan. 1994.*

[MCINTYRE85] McIntyre D.R., Pechura M.A.; “Data Compression using Static Huffman Code-Decode Tables”, *Journal of the ACM 28:6,612-616.*

[MOFF95] Moffat, A., Zobel, J.; “Information retrieval systems for large document collections”, *Text Retrieval Conference (TREC-3), pp. 85-93. Gaithersburg, MD, USA, 2-4 Nov. 1994.*

[PIKE81] Pike J.; "Text Compression using a 4-bit Coding Scheme",
The Computer Journal 24:4, 324-330.

[STORER87] Storer J.A., Tsang S.K.; “Data Compression Experiments Using Static and Dynamic Dictionaries”, *Technical Report CS-84-118, CS Dept., Brandeis University Waltham, MA*

[WAGNER73] Wagner R.A.; “Common Phrases and Minimum-Space Text Storage”,
Communications of the ACM 16:3, 148-152.

[WITT94] Witten, I., Moffat, A., Bell, T.; “Managing Gigabytes”,
Van Nostrand Reinhold, 1994.

[YANNAKOUDAKIS82] Yannakoudakis E.J., Goyal P., Huggil J.A.; ‘The Generation and Use of Text Fragments for Data Compression”, *Information Processing and Management 18:1, pp. 15-21.*

[YIAN92] P. Yianilos; “Data structures and algorithms for nearest neighbor search in general metric spaces”, *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 311-321, Orlando, Fla., 1992.*

[ZOBE95] Zobel, J., Moffat, A.; “Adding compression to a full-text retrieval system”,
Software - Practice and Experience (Aug. 1995) vol.25, no.8, pp. 891-903.

NOTES

NOTES

NOTES

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Javan and C++, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as ten technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates and Prentice Hall PTR.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Special subscription rate for *The Linux Journal.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Apunix Computer Services
Auspex Systems, Inc.
Boeing Commercial
Crosswind Technologies, Inc.
Digital Equipment Corporation
Earthlink Network, Inc.
Invincible Technologies
Lucent Technologies, Bell Labs
Motorola Research & Development
MTI Technology Corporation
Nimrod AS
O'Reilly & Associates
Sun Microsystems, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Digital Equipment Corporation
ESM Services, Inc.
Global Networking and Computing, Inc.
Great Circle Associates
OnLine Staffing
Sprint Paranet
Texas Instruments, Inc.
TransQuest Technologies, Inc.
UNIX Guru Universe

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-91-X